

Name: Devan Patel

UCID: ddp47

Email: ddp47@njit.edu

Option #: 1

Algorithms: Category 2 - Random Forests (Random Forest Classifier), Category 5 - Naïve Bayes (Naïve Bayes Classifier)

Tools: CPython, Jupyter Notebook, Pandas, Numpy, Matplotlib, Seaborn, Sklearn

Dataset: <https://www.kaggle.com/uciml/mushroom-classification>

Software: *source code of classification algorithms and websites where the software can be downloaded*

https://github.com/scikit-learn/scikit-learn/blob/master/sklearn/ensemble/_forest.py

```
"""
Forest of trees-based ensemble methods.

Those methods include random forests and extremely randomized trees.

The module structure is the following:

- The ``BaseForest`` base class implements a common ``fit`` method for all
  the estimators in the module. The ``fit`` method of the base ``Forest``
  class calls the ``fit`` method of each sub-estimator on random samples
  (with replacement, a.k.a. bootstrap) of the training set.

  The init of the sub-estimator is further delegated to the
  ``BaseEnsemble`` constructor.

- The ``ForestClassifier`` and ``ForestRegressor`` base classes further
  implement the prediction logic by computing an average of the predicted
  outcomes of the sub-estimators.

- The ``RandomForestClassifier`` and ``RandomForestRegressor`` derived
  classes provide the user with concrete implementations of
  the forest ensemble method using classical, deterministic
  ``DecisionTreeClassifier`` and ``DecisionTreeRegressor`` as
  sub-estimator implementations.

- The ``ExtraTreesClassifier`` and ``ExtraTreesRegressor`` derived
  classes provide the user with concrete implementations of the
  forest ensemble method using the extremely randomized trees
  ``ExtraTreeClassifier`` and ``ExtraTreeRegressor`` as
  sub-estimator implementations.

Single and multi-output problems are both handled.
"""

# Authors: Gilles Louppe <g.louppe@gmail.com>
#          Brian Holt <bdholt1@gmail.com>
#          Joly Arnaud <arnaud.v.joly@gmail.com>
#          Fares Hedayati <fares.hedayati@gmail.com>
```

```

# License: BSD 3 clause

import numbers
from warnings import catch_warnings, simplefilter, warn
import threading

from abc import ABCMeta, abstractmethod
import numpy as np
from scipy.sparse import issparse
from scipy.sparse import hstack as sparse_hstack
from joblib import Parallel, delayed

from ..base import ClassifierMixin, RegressorMixin, MultiOutputMixin
from ..metrics import r2_score
from ..preprocessing import OneHotEncoder
from ..tree import (DecisionTreeClassifier, DecisionTreeRegressor,
                    ExtraTreeClassifier, ExtraTreeRegressor)
from ..tree._tree import DTYPES, DOUBLE
from ..utils import check_random_state, check_array, compute_sample_weight
from ..exceptions import DataConversionWarning
from .._base import BaseEnsemble, _partition_estimators
from ..utils.fixes import _joblib_parallel_args
from ..utils.multiclass import check_classification_targets
from ..utils.validation import check_is_fitted, _check_sample_weight
from ..utils.validation import _deprecate_positional_args

__all__ = ["RandomForestClassifier",
           "RandomForestRegressor",
           "ExtraTreesClassifier",
           "ExtraTreesRegressor",
           "RandomTreesEmbedding"]

MAX_INT = np.iinfo(np.int32).max

def _get_n_samples_bootstrap(n_samples, max_samples):
    """
    Get the number of samples in a bootstrap sample.

    Parameters
    -----
    n_samples : int
        Number of samples in the dataset.
    max_samples : int or float
        The maximum number of samples to draw from the total available:
        - if float, this indicates a fraction of the total and should be
          the interval `(0, 1)`;
        - if int, this indicates the exact number of samples;
        - if None, this indicates the total number of samples.

    Returns
    -----
    n_samples_bootstrap : int
        The total number of samples to draw for the bootstrap sample.
    """
    if max_samples is None:
        return n_samples

    if isinstance(max_samples, numbers.Integral):
        if not (1 <= max_samples <= n_samples):
            msg = "`max_samples` must be in range 1 to {} but got value {}"

```

```

        raise ValueError(msg.format(n_samples, max_samples))
    return max_samples

if isinstance(max_samples, numbers.Real):
    if not (0 < max_samples < 1):
        msg = "`max_samples` must be in range (0, 1) but got value {}"
        raise ValueError(msg.format(max_samples))
    return int(round(n_samples * max_samples))

msg = "`max_samples` should be int or float, but got type '{}'"
raise TypeError(msg.format(type(max_samples)))

def _generate_sample_indices(random_state, n_samples, n_samples_bootstrap):
    """
    Private function used to _parallel_build_trees function."""

    random_instance = check_random_state(random_state)
    sample_indices = random_instance.randint(0, n_samples, n_samples_bootstrap)

    return sample_indices

def _generate_unsampled_indices(random_state, n_samples, n_samples_bootstrap):
    """
    Private function used to forest._set_oob_score function."""

    sample_indices = _generate_sample_indices(random_state, n_samples,
                                              n_samples_bootstrap)
    sample_counts = np.bincount(sample_indices, minlength=n_samples)
    unsampled_mask = sample_counts == 0
    indices_range = np.arange(n_samples)
    unsampled_indices = indices_range[unsampled_mask]

    return unsampled_indices

def _parallel_build_trees(tree, forest, X, y, sample_weight, tree_idx, n_trees,
                        verbose=0, class_weight=None,
                        n_samples_bootstrap=None):
    """
    Private function used to fit a single tree in parallel."""

    if verbose > 1:
        print("building tree %d of %d" % (tree_idx + 1, n_trees))

    if forest.bootstrap:
        n_samples = X.shape[0]
        if sample_weight is None:
            curr_sample_weight = np.ones((n_samples,), dtype=np.float64)
        else:
            curr_sample_weight = sample_weight.copy()

        indices = _generate_sample_indices(tree.random_state, n_samples,
                                          n_samples_bootstrap)
        sample_counts = np.bincount(indices, minlength=n_samples)
        curr_sample_weight *= sample_counts

        if class_weight == 'subsample':
            with catch_warnings():
                simplefilter('ignore', DeprecationWarning)
                curr_sample_weight *= compute_sample_weight('auto', y, indices)
        elif class_weight == 'balanced_subsample':
            curr_sample_weight *= compute_sample_weight('balanced', y, indices)

```

```

        tree.fit(X, y, sample_weight=curr_sample_weight, check_input=False)
    else:
        tree.fit(X, y, sample_weight=sample_weight, check_input=False)

    return tree

class BaseForest(MultiOutputMixin, BaseEnsemble, metaclass=ABCMeta):
    """
    Base class for forests of trees.

    Warning: This class should not be used directly. Use derived classes
    instead.
    """

    @abstractmethod
    def __init__(self,
                 base_estimator,
                 n_estimators=100, *,
                 estimator_params=tuple(),
                 bootstrap=False,
                 oob_score=False,
                 n_jobs=None,
                 random_state=None,
                 verbose=0,
                 warm_start=False,
                 class_weight=None,
                 max_samples=None):
        super().__init__(
            base_estimator=base_estimator,
            n_estimators=n_estimators,
            estimator_params=estimator_params)

        self.bootstrap = bootstrap
        self.oob_score = oob_score
        self.n_jobs = n_jobs
        self.random_state = random_state
        self.verbose = verbose
        self.warm_start = warm_start
        self.class_weight = class_weight
        self.max_samples = max_samples

    def apply(self, X):
        """
        Apply trees in the forest to X, return leaf indices.

        Parameters
        -----
        X : {array-like, sparse matrix} of shape (n_samples, n_features)
            The input samples. Internally, its dtype will be converted to
            ``dtype=np.float32``. If a sparse matrix is provided, it will be
            converted into a sparse ``csr_matrix``.

        Returns
        -----
        X_leaves : ndarray of shape (n_samples, n_estimators)
            For each datapoint x in X and for each tree in the forest,
            return the index of the leaf x ends up in.
        """
        X = self._validate_X_predict(X)
        results = Parallel(n_jobs=self.n_jobs, verbose=self.verbose,
                           **_joblib_parallel_args(prefer="threads"))(
            delayed(tree.apply)(X, check_input=False)

```

```

        for tree in self.estimators_)

    return np.array(results).T

def decision_path(self, X):
    """
    Return the decision path in the forest.

    .. versionadded:: 0.18

    Parameters
    -----
    X : {array-like, sparse matrix} of shape (n_samples, n_features)
        The input samples. Internally, its dtype will be converted to
        ``dtype=np.float32``. If a sparse matrix is provided, it will be
        converted into a sparse ``csr_matrix``.

    Returns
    -----
    indicator : sparse matrix of shape (n_samples, n_nodes)
        Return a node indicator matrix where non zero elements indicates
        that the samples goes through the nodes. The matrix is of CSR
        format.

    n_nodes_ptr : ndarray of shape (n_estimators + 1,)
        The columns from indicator[n_nodes_ptr[i]:n_nodes_ptr[i+1]]
        gives the indicator value for the i-th estimator.

    """
    X = self._validate_X_predict(X)
    indicators = Parallel(n_jobs=self.n_jobs, verbose=self.verbose,
                          **_joblib_parallel_args(prefer='threads'))(
        delayed(tree.decision_path)(X, check_input=False)
        for tree in self.estimators_)

    n_nodes = [0]
    n_nodes.extend([i.shape[1] for i in indicators])
    n_nodes_ptr = np.array(n_nodes).cumsum()

    return sparse_hstack(indicators).tocsr(), n_nodes_ptr

def fit(self, X, y, sample_weight=None):
    """
    Build a forest of trees from the training set (X, y).

    Parameters
    -----
    X : {array-like, sparse matrix} of shape (n_samples, n_features)
        The training input samples. Internally, its dtype will be converted
        to ``dtype=np.float32``. If a sparse matrix is provided, it will be
        converted into a sparse ``csc_matrix``.

    y : array-like of shape (n_samples,) or (n_samples, n_outputs)
        The target values (class labels in classification, real numbers in
        regression).

    sample_weight : array-like of shape (n_samples,), default=None
        Sample weights. If None, then samples are equally weighted. Splits
        that would create child nodes with net zero or negative weight are
        ignored while searching for a split in each node. In the case of
        classification, splits are also ignored if they would result in any
        single class carrying a negative weight in either child node.

```

```

>Returns
-----
self : object
"""
# Validate or convert input data
if issparse(y):
    raise ValueError(
        "sparse multilabel-indicator for y is not supported."
    )
X, y = self._validate_data(X, y, multi_output=True,
                           accept_sparse="csc", dtype=DTYPE)
if sample_weight is not None:
    sample_weight = _check_sample_weight(sample_weight, X)

if issparse(X):
    # Pre-sort indices to avoid that each individual tree of the
    # ensemble sorts the indices.
    X.sort_indices()

# Remap output
self.n_features_ = X.shape[1]

y = np.atleast_1d(y)
if y.ndim == 2 and y.shape[1] == 1:
    warn("A column-vector y was passed when a 1d array was"
         " expected. Please change the shape of y to "
         "(n_samples,), for example using ravel().",
         DataConversionWarning, stacklevel=2)

if y.ndim == 1:
    # reshape is necessary to preserve the data contiguity against vs
    # [:, np.newaxis] that does not.
    y = np.reshape(y, (-1, 1))

self.n_outputs_ = y.shape[1]

y, expanded_class_weight = self._validate_y_class_weight(y)

if getattr(y, "dtype", None) != DOUBLE or not y.flags.contiguous:
    y = np.ascontiguousarray(y, dtype=DOUBLE)

if expanded_class_weight is not None:
    if sample_weight is not None:
        sample_weight = sample_weight * expanded_class_weight
    else:
        sample_weight = expanded_class_weight

# Get bootstrap sample size
n_samples_bootstrap = _get_n_samples_bootstrap(
    n_samples=X.shape[0],
    max_samples=self.max_samples
)

# Check parameters
self._validate_estimator()

if not self.bootstrap and self.oob_score:
    raise ValueError("Out of bag estimation only available"
                     " if bootstrap=True")

random_state = check_random_state(self.random_state)

if not self.warm_start or not hasattr(self, "estimators_"):

```

```

        # Free allocated memory, if any
        self.estimators_ = []

    n_more_estimators = self.n_estimators - len(self.estimators_)

    if n_more_estimators < 0:
        raise ValueError('n_estimators=%d must be larger or equal to '
                         'len(estimators_)=%d when warm_start==True'
                         % (self.n_estimators, len(self.estimators_)))

    elif n_more_estimators == 0:
        warn("Warm-start fitting without increasing n_estimators does not "
             "fit new trees.")
    else:
        if self.warm_start and len(self.estimators_) > 0:
            # We draw from the random state to get the random state we
            # would have got if we hadn't used a warm_start.
            random_state.randint(MAX_INT, size=len(self.estimators_))

        trees = [self._make_estimator(append=False,
                                      random_state=random_state)
                 for i in range(n_more_estimators)]

        # Parallel loop: we prefer the threading backend as the Cython code
        # for fitting the trees is internally releasing the Python GIL
        # making threading more efficient than multiprocessing in
        # that case. However, for joblib 0.12+ we respect any
        # parallel_backend contexts set at a higher level,
        # since correctness does not rely on using threads.
        trees = Parallel(n_jobs=self.n_jobs, verbose=self.verbose,
                         **joblib_parallel_args(prefer='threads'))(
            delayed(_parallel_build_trees)(
                t, self, X, y, sample_weight, i, len(trees),
                verbose=self.verbose, class_weight=self.class_weight,
                n_samples_bootstrap=n_samples_bootstrap)
            for i, t in enumerate(trees))

        # Collect newly grown trees
        self.estimators_.extend(trees)

    if self.oob_score:
        self._set_oob_score(X, y)

    # Decapsulate classes_ attributes
    if hasattr(self, "classes_") and self.n_outputs_ == 1:
        self.n_classes_ = self.classes_[0]
        self.classes_ = self.classes_[0]

    return self

@abstractmethod
def _set_oob_score(self, X, y):
    """
    Calculate out of bag predictions and score."""
    pass

def _validate_y_class_weight(self, y):
    # Default implementation
    return y, None

def _validate_X_predict(self, X):
    """
    Validate X whenever one tries to predict, apply, predict_proba."""
    check_is_fitted(self)

```

```

        return self.estimators_[0]._validate_X_predict(X, check_input=True)

@property
def feature_importances_(self):
    """
    The impurity-based feature importances.

    The higher, the more important the feature.
    The importance of a feature is computed as the (normalized)
    total reduction of the criterion brought by that feature. It is also
    known as the Gini importance.

    Warning: impurity-based feature importances can be misleading for
    high cardinality features (many unique values). See
    :func:`sklearn.inspection.permutation_importance` as an alternative.

    Returns
    ------
    feature_importances_ : ndarray of shape (n_features,)
        The values of this array sum to 1, unless all trees are single node
        trees consisting of only the root node, in which case it will be an
        array of zeros.
    """
    check_is_fitted(self)

    all_importances = Parallel(n_jobs=self.n_jobs,
                               **_joblib_parallel_args(prefer='threads'))(
        delayed(getattr)(tree, 'feature_importances_')
        for tree in self.estimators_ if tree.tree_.node_count > 1)

    if not all_importances:
        return np.zeros(self.n_features_, dtype=np.float64)

    all_importances = np.mean(all_importances,
                             axis=0, dtype=np.float64)
    return all_importances / np.sum(all_importances)

def _accumulate_prediction(predict, X, out, lock):
    """
    This is a utility function for joblib's Parallel.

    It can't go locally in ForestClassifier or ForestRegressor, because joblib
    complains that it cannot pickle it when placed there.
    """
    prediction = predict(X, check_input=False)
    with lock:
        if len(out) == 1:
            out[0] += prediction
        else:
            for i in range(len(out)):
                out[i] += prediction[i]

class ForestClassifier(ClassifierMixin, BaseForest, metaclass=ABCMeta):
    """
    Base class for forest of trees-based classifiers.

    Warning: This class should not be used directly. Use derived classes
    instead.
    """

```

```

@abstractmethod
def __init__(self,
             base_estimator,
             n_estimators=100, *,
             estimator_params=tuple(),
             bootstrap=False,
             oob_score=False,
             n_jobs=None,
             random_state=None,
             verbose=0,
             warm_start=False,
             class_weight=None,
             max_samples=None):
    super().__init__(
        base_estimator,
        n_estimators=n_estimators,
        estimator_params=estimator_params,
        bootstrap=bootstrap,
        oob_score=oob_score,
        n_jobs=n_jobs,
        random_state=random_state,
        verbose=verbose,
        warm_start=warm_start,
        class_weight=class_weight,
        max_samples=max_samples)

def _set_oob_score(self, X, y):
    """
    Compute out-of-bag score."""
    X = check_array(X, dtype=DTYPE, accept_sparse='csr')

    n_classes_ = self.n_classes_
    n_samples = y.shape[0]

    oob_decision_function = []
    oob_score = 0.0
    predictions = [np.zeros((n_samples, n_classes_[k]))
                  for k in range(self.n_outputs_)]
    n_samples_bootstrap = _get_n_samples_bootstrap(
        n_samples, self.max_samples
    )

    for estimator in self.estimators_:
        unsampled_indices = _generate_unsampled_indices(
            estimator.random_state, n_samples, n_samples_bootstrap)
        p_estimator = estimator.predict_proba(X[unsampled_indices, :],
                                              check_input=False)

        if self.n_outputs_ == 1:
            p_estimator = [p_estimator]

        for k in range(self.n_outputs_):
            predictions[k][unsampled_indices, :] += p_estimator[k]

    for k in range(self.n_outputs_):
        if (predictions[k].sum(axis=1) == 0).any():
            warn("Some inputs do not have OOB scores. "
                 "This probably means too few trees were used "
                 "to compute any reliable oob estimates.")

    decision = (predictions[k] /
                predictions[k].sum(axis=1)[:, np.newaxis])

```

```

        oob_decision_function.append(decision)
        oob_score += np.mean(y[:, k] ==
                             np.argmax(predictions[k], axis=1), axis=0)

    if self.n_outputs_ == 1:
        self.oob_decision_function_ = oob_decision_function[0]
    else:
        self.oob_decision_function_ = oob_decision_function

    self.oob_score_ = oob_score / self.n_outputs_

def _validate_y_class_weight(self, y):
    check_classification_targets(y)

    y = np.copy(y)
    expanded_class_weight = None

    if self.class_weight is not None:
        y_original = np.copy(y)

    self.classes_ = []
    self.n_classes_ = []

    y_store_unique_indices = np.zeros(y.shape, dtype=np.int)
    for k in range(self.n_outputs_):
        classes_k, y_store_unique_indices[:, k] = \
            np.unique(y[:, k], return_inverse=True)
        self.classes_.append(classes_k)
        self.n_classes_.append(classes_k.shape[0])
    y = y_store_unique_indices

    if self.class_weight is not None:
        valid_presets = ('balanced', 'balanced_subsample')
        if isinstance(self.class_weight, str):
            if self.class_weight not in valid_presets:
                raise ValueError('Valid presets for class_weight include '
                                 '"balanced" and "balanced_subsample".'
                                 'Given "%s".'
                                 % self.class_weight)
        if self.warm_start:
            warn('class_weight presets "balanced" or '
                 '"balanced_subsample" are '
                 'not recommended for warm_start if the fitted data '
                 'differs from the full dataset. In order to use '
                 '"balanced" weights, use compute_class_weight '
                 '(["balanced", classes, y]). In place of y you can use '
                 'a large enough sample of the full training set '
                 'target to properly estimate the class frequency '
                 'distributions. Pass the resulting weights as the '
                 'class_weight parameter.')
    if (self.class_weight != 'balanced_subsample' or
        not self.bootstrap):
        if self.class_weight == "balanced_subsample":
            class_weight = "balanced"
        else:
            class_weight = self.class_weight
            expanded_class_weight = compute_sample_weight(class_weight,
                                                          y_original)

    return y, expanded_class_weight

def predict(self, X):

```

```

"""
Predict class for X.

The predicted class of an input sample is a vote by the trees in
the forest, weighted by their probability estimates. That is,
the predicted class is the one with highest mean probability
estimate across the trees.

Parameters
-----
X : {array-like, sparse matrix} of shape (n_samples, n_features)
    The input samples. Internally, its dtype will be converted to
    ``dtype=np.float32``. If a sparse matrix is provided, it will be
    converted into a sparse ``csr_matrix``.

Returns
-----
y : ndarray of shape (n_samples,) or (n_samples, n_outputs)
    The predicted classes.
"""

proba = self.predict_proba(X)

if self.n_outputs_ == 1:
    return self.classes_.take(np.argmax(proba, axis=1), axis=0)

else:
    n_samples = proba[0].shape[0]
    # all dtypes should be the same, so just take the first
    class_type = self.classes_[0].dtype
    predictions = np.empty((n_samples, self.n_outputs_),
                           dtype=class_type)

    for k in range(self.n_outputs_):
        predictions[:, k] = self.classes_[k].take(np.argmax(proba[k],
                                              axis=1),
                                              axis=0)

    return predictions

def predict_proba(self, X):
    """
    Predict class probabilities for X.

    The predicted class probabilities of an input sample are computed as
    the mean predicted class probabilities of the trees in the forest.
    The class probability of a single tree is the fraction of samples of
    the same class in a leaf.

    Parameters
    -----
    X : {array-like, sparse matrix} of shape (n_samples, n_features)
        The input samples. Internally, its dtype will be converted to
        ``dtype=np.float32``. If a sparse matrix is provided, it will be
        converted into a sparse ``csr_matrix``.

    Returns
    -----
    p : ndarray of shape (n_samples, n_classes), or a list of n_outputs
        such arrays if n_outputs > 1.
        The class probabilities of the input samples. The order of the
        classes corresponds to that in the attribute :term:`classes_`.

    """
    check_is_fitted(self)

```

```

# Check data
X = self._validate_X_predict(X)

# Assign chunk of trees to jobs
n_jobs, _, _ = _partition_estimators(self.n_estimators, self.n_jobs)

# avoid storing the output of every estimator by summing them here
all_proba = [np.zeros((X.shape[0], j), dtype=np.float64)
             for j in np.atleast_1d(self.n_classes_)]
lock = threading.Lock()
Parallel(n_jobs=n_jobs, verbose=self.verbose,
          **_joblib_parallel_args(require="sharedmem"))(
    delayed(_accumulate_prediction)(e.predict_proba, X, all_proba,
                                     lock)
    for e in self.estimators_)

for proba in all_proba:
    proba /= len(self.estimators_)

if len(all_proba) == 1:
    return all_proba[0]
else:
    return all_proba

def predict_log_proba(self, X):
    """
    Predict class log-probabilities for X.

    The predicted class log-probabilities of an input sample is computed as
    the log of the mean predicted class probabilities of the trees in the
    forest.

    Parameters
    -----
    X : {array-like, sparse matrix} of shape (n_samples, n_features)
        The input samples. Internally, its dtype will be converted to
        ``dtype=np.float32``. If a sparse matrix is provided, it will be
        converted into a sparse ``csr_matrix``.

    Returns
    -----
    p : ndarray of shape (n_samples, n_classes), or a list of n_outputs
        such arrays if n_outputs > 1.
        The class probabilities of the input samples. The order of the
        classes corresponds to that in the attribute :term:`classes_`.
    """
    proba = self.predict_proba(X)

    if self.n_outputs_ == 1:
        return np.log(proba)

    else:
        for k in range(self.n_outputs_):
            proba[k] = np.log(proba[k])

    return proba

class ForestRegressor(RegressorMixin, BaseForest, metaclass=ABCMeta):
    """
    Base class for forest of trees-based regressors.

    Warning: This class should not be used directly. Use derived classes
    """

```

```

instead.
"""

@abstractmethod
def __init__(self,
             base_estimator,
             n_estimators=100, *,
             estimator_params=tuple(),
             bootstrap=False,
             oob_score=False,
             n_jobs=None,
             random_state=None,
             verbose=0,
             warm_start=False,
             max_samples=None):
    super().__init__(
        base_estimator,
        n_estimators=n_estimators,
        estimator_params=estimator_params,
        bootstrap=bootstrap,
        oob_score=oob_score,
        n_jobs=n_jobs,
        random_state=random_state,
        verbose=verbose,
        warm_start=warm_start,
        max_samples=max_samples)

def predict(self, X):
    """
    Predict regression target for X.

    The predicted regression target of an input sample is computed as the
    mean predicted regression targets of the trees in the forest.

    Parameters
    -----
    X : {array-like, sparse matrix} of shape (n_samples, n_features)
        The input samples. Internally, its dtype will be converted to
        ``dtype=np.float32``. If a sparse matrix is provided, it will be
        converted into a sparse ``csr_matrix``.

    Returns
    -----
    y : ndarray of shape (n_samples,) or (n_samples, n_outputs)
        The predicted values.
    """
    check_is_fitted(self)
    # Check data
    X = self._validate_X_predict(X)

    # Assign chunk of trees to jobs
    n_jobs, _, _ = _partition_estimators(self.n_estimators, self.n_jobs)

    # avoid storing the output of every estimator by summing them here
    if self.n_outputs_ > 1:
        y_hat = np.zeros((X.shape[0], self.n_outputs_), dtype=np.float64)
    else:
        y_hat = np.zeros((X.shape[0]), dtype=np.float64)

    # Parallel loop
    lock = threading.Lock()
    Parallel(n_jobs=n_jobs, verbose=self.verbose,
              **joblib_parallel_args(require="sharedmem"))(

```

```

        delayed(_accumulate_prediction)(e.predict, X, [y_hat], lock)
        for e in self.estimators_)

    y_hat /= len(self.estimators_)

    return y_hat

def _set_oob_score(self, X, y):
    """
    Compute out-of-bag scores."""
    X = check_array(X, dtype=DTYPE, accept_sparse='csr')

    n_samples = y.shape[0]

    predictions = np.zeros((n_samples, self.n_outputs_))
    n_predictions = np.zeros((n_samples, self.n_outputs_))

    n_samples_bootstrap = _get_n_samples_bootstrap(
        n_samples, self.max_samples
    )

    for estimator in self.estimators_:
        unsampled_indices = _generate_unsampled_indices(
            estimator.random_state, n_samples, n_samples_bootstrap)
        p_estimator = estimator.predict(
            X[unsampled_indices, :], check_input=False)

        if self.n_outputs_ == 1:
            p_estimator = p_estimator[:, np.newaxis]

        predictions[unsampled_indices, :] += p_estimator
        n_predictions[unsampled_indices, :] += 1

    if (n_predictions == 0).any():
        warn("Some inputs do not have OOB scores. "
             "This probably means too few trees were used "
             "to compute any reliable oob estimates.")
        n_predictions[n_predictions == 0] = 1

    predictions /= n_predictions
    self.oob_prediction_ = predictions

    if self.n_outputs_ == 1:
        self.oob_prediction_ = \
            self.oob_prediction_.reshape((n_samples,))

    self.oob_score_ = 0.0

    for k in range(self.n_outputs_):
        self.oob_score_ += r2_score(y[:, k],
                                    predictions[:, k])

    self.oob_score_ /= self.n_outputs_

def _compute_partial_dependence_recursion(self, grid, target_features):
    """
    Fast partial dependence computation.

    Parameters
    -----
    grid : ndarray of shape (n_samples, n_target_features)
        The grid points on which the partial dependence should be
        evaluated.
    target_features : ndarray of shape (n_target_features)

```

```

    The set of target features for which the partial dependence
    should be evaluated.

Returns
-----
averaged_predictions : ndarray of shape (n_samples,)
    The value of the partial dependence function on each grid point.
"""
grid = np.asarray(grid, dtype=DTYPE, order='C')
averaged_predictions = np.zeros(shape=grid.shape[0],
                                 dtype=np.float64, order='C')

for tree in self.estimators_:
    # Note: we don't sum in parallel because the GIL isn't released in
    # the fast method.
    tree.tree_.compute_partial_dependence(
        grid, target_features, averaged_predictions)
# Average over the forest
averaged_predictions /= len(self.estimators_)

return averaged_predictions

class RandomForestClassifier(ForestClassifier):
"""
A random forest classifier.

A random forest is a meta estimator that fits a number of decision tree
classifiers on various sub-samples of the dataset and uses averaging to
improve the predictive accuracy and control over-fitting.
The sub-sample size is controlled with the `max_samples` parameter if
`bootstrap=True` (default), otherwise the whole dataset is used to build
each tree.

Read more in the :ref:`User Guide <forest>`.

Parameters
-----
n_estimators : int, default=100
    The number of trees in the forest.

    .. versionchanged:: 0.22
        The default value of ``n_estimators`` changed from 10 to 100
        in 0.22.

criterion : {"gini", "entropy"}, default="gini"
    The function to measure the quality of a split. Supported criteria are
    "gini" for the Gini impurity and "entropy" for the information gain.
    Note: this parameter is tree-specific.

max_depth : int, default=None
    The maximum depth of the tree. If None, then nodes are expanded until
    all leaves are pure or until all leaves contain less than
    min_samples_split samples.

min_samples_split : int or float, default=2
    The minimum number of samples required to split an internal node:

    - If int, then consider `min_samples_split` as the minimum number.
    - If float, then `min_samples_split` is a fraction and
      `ceil(min_samples_split * n_samples)` are the minimum
      number of samples for each split.

    .. versionchanged:: 0.18

```

```
Added float values for fractions.
```

```
min_samples_leaf : int or float, default=1
The minimum number of samples required to be at a leaf node.
A split point at any depth will only be considered if it leaves at
least ``min_samples_leaf`` training samples in each of the left and
right branches. This may have the effect of smoothing the model,
especially in regression.
```

- If int, then consider `min_samples_leaf` as the minimum number.
- If float, then `min_samples_leaf` is a fraction and
`ceil(min_samples_leaf * n_samples)` are the minimum
number of samples for each node.

```
.. versionchanged:: 0.18
    Added float values for fractions.
```

```
min_weight_fraction_leaf : float, default=0.0
The minimum weighted fraction of the sum total of weights (of all
the input samples) required to be at a leaf node. Samples have
equal weight when sample_weight is not provided.
```

```
max_features : {"auto", "sqrt", "log2"}, int or float, default="auto"
The number of features to consider when looking for the best split:
```

- If int, then consider `max_features` features at each split.
- If float, then `max_features` is a fraction and
`int(max_features * n_features)` features are considered at each
split.
- If "auto", then `max_features=sqrt(n_features)`.
- If "sqrt", then `max_features=sqrt(n_features)` (same as "auto").
- If "log2", then `max_features=log2(n_features)`.
- If None, then `max_features=n_features`.

Note: the search for a split does not stop until at least one valid partition of the node samples is found, even if it requires to effectively inspect more than ``max_features`` features.

```
max_leaf_nodes : int, default=None
Grow trees with ``max_leaf_nodes`` in best-first fashion.
Best nodes are defined as relative reduction in impurity.
If None then unlimited number of leaf nodes.
```

```
min_impurity_decrease : float, default=0.0
A node will be split if this split induces a decrease of the impurity
greater than or equal to this value.
```

The weighted impurity decrease equation is the following::

$$N_t / N * (\text{impurity} - N_{t,R} / N_t * \text{right_impurity} - N_{t,L} / N_t * \text{left_impurity})$$

where ``N`` is the total number of samples, ``N_t`` is the number of samples at the current node, ``N_{t,L}`` is the number of samples in the left child, and ``N_{t,R}`` is the number of samples in the right child.

``N``, ``N_t``, ``N_{t,R}`` and ``N_{t,L}`` all refer to the weighted sum, if ``sample_weight`` is passed.

```
.. versionadded:: 0.19
```

```
min_impurity_split : float, default=None
Threshold for early stopping in tree growth. A node will split
```

```

if its impurity is above the threshold, otherwise it is a leaf.

.. deprecated:: 0.19
    ``min_impurity_split`` has been deprecated in favor of
    ``min_impurity_decrease`` in 0.19. The default value of
    ``min_impurity_split`` has changed from 1e-7 to 0 in 0.23 and it
    will be removed in 0.25. Use ``min_impurity_decrease`` instead.

bootstrap : bool, default=True
    Whether bootstrap samples are used when building trees. If False, the
    whole dataset is used to build each tree.

oob_score : bool, default=False
    Whether to use out-of-bag samples to estimate
    the generalization accuracy.

n_jobs : int, default=None
    The number of jobs to run in parallel. :meth:`fit`, :meth:`predict`,
    :meth:`decision_path` and :meth:`apply` are all parallelized over the
    trees. ``None`` means 1 unless in a :obj:`joblib.parallel_backend`
    context. ``-1`` means using all processors. See :term:`Glossary
    <n_jobs>` for more details.

random_state : int or RandomState, default=None
    Controls both the randomness of the bootstrapping of the samples used
    when building trees (if ``bootstrap=True``) and the sampling of the
    features to consider when looking for the best split at each node
    (if ``max_features < n_features``).
    See :term:`Glossary <random_state>` for details.

verbose : int, default=0
    Controls the verbosity when fitting and predicting.

warm_start : bool, default=False
    When set to ``True``, reuse the solution of the previous call to fit
    and add more estimators to the ensemble, otherwise, just fit a whole
    new forest. See :term:`the Glossary <warm_start>`.

class_weight : {"balanced", "balanced_subsample"}, dict or list of dicts, \
    default=None
    Weights associated with classes in the form ``{class_label: weight}``.
    If not given, all classes are supposed to have weight one. For
    multi-output problems, a list of dicts can be provided in the same
    order as the columns of y.

    Note that for multioutput (including multilabel) weights should be
    defined for each class of every column in its own dict. For example,
    for four-class multilabel classification weights should be
    [{0: 1, 1: 1}, {0: 1, 1: 5}, {0: 1, 1: 1}, {0: 1, 1: 1}] instead of
    [{1:1}, {2:5}, {3:1}, {4:1}].
```

The "balanced" mode uses the values of y to automatically adjust weights inversely proportional to class frequencies in the input data as ``n_samples / (n_classes * np.bincount(y))``

The "balanced_subsample" mode is the same as "balanced" except that weights are computed based on the bootstrap sample for every tree grown.

For multi-output, the weights of each column of y will be multiplied.

Note that these weights will be multiplied with sample_weight (passed

```
through the fit method) if sample_weight is specified.

ccp_alpha : non-negative float, default=0.0
    Complexity parameter used for Minimal Cost-Complexity Pruning. The
    subtree with the largest cost complexity that is smaller than
    ``ccp_alpha`` will be chosen. By default, no pruning is performed. See
    :ref:`minimal_cost_complexity_pruning` for details.

.. versionadded:: 0.22

max_samples : int or float, default=None
    If bootstrap is True, the number of samples to draw from X
    to train each base estimator.

    - If None (default), then draw `X.shape[0]` samples.
    - If int, then draw `max_samples` samples.
    - If float, then draw `max_samples * X.shape[0]` samples. Thus,
      `max_samples` should be in the interval `(0, 1)`.

.. versionadded:: 0.22

Attributes
-----
base_estimator_ : DecisionTreeClassifier
    The child estimator template used to create the collection of fitted
    sub-estimators.

estimators_ : list of DecisionTreeClassifier
    The collection of fitted sub-estimators.

classes_ : ndarray of shape (n_classes,) or a list of such arrays
    The classes labels (single output problem), or a list of arrays of
    class labels (multi-output problem).

n_classes_ : int or list
    The number of classes (single output problem), or a list containing the
    number of classes for each output (multi-output problem).

n_features_ : int
    The number of features when ``fit`` is performed.

n_outputs_ : int
    The number of outputs when ``fit`` is performed.

feature_importances_ : ndarray of shape (n_features,)
    The impurity-based feature importances.
    The higher, the more important the feature.
    The importance of a feature is computed as the (normalized)
    total reduction of the criterion brought by that feature. It is also
    known as the Gini importance.

    Warning: impurity-based feature importances can be misleading for
    high cardinality features (many unique values). See
    :func:`sklearn.inspection.permutation_importance` as an alternative.

oob_score_ : float
    Score of the training dataset obtained using an out-of-bag estimate.
    This attribute exists only when ``oob_score`` is True.

oob_decision_function_ : ndarray of shape (n_samples, n_classes)
    Decision function computed with out-of-bag estimate on the training
    set. If n_estimators is small it might be possible that a data point
    was never left out during the bootstrap. In this case,
```

```
`oob_decision_function_` might contain NaN. This attribute exists  
only when ``oob_score`` is True.
```

See Also

`DecisionTreeClassifier`, `ExtraTreesClassifier`

Notes

The default values for the parameters controlling the size of the trees
(e.g. ``max_depth``, ``min_samples_leaf``, etc.) lead to fully grown and
unpruned trees which can potentially be very large on some data sets. To
reduce memory consumption, the complexity and size of the trees should be
controlled by setting those parameter values.

The features are always randomly permuted at each split. Therefore,
the best found split may vary, even with the same training data,
``max_features=n_features`` and ``bootstrap=False``, if the improvement
of the criterion is identical for several splits enumerated during the
search of the best split. To obtain a deterministic behaviour during
fitting, ``random_state`` has to be fixed.

References

.. [1] L. Breiman, "Random Forests", *Machine Learning*, 45(1), 5-32, 2001.

Examples

```
>>> from sklearn.ensemble import RandomForestClassifier  
>>> from sklearn.datasets import make_classification  
>>> X, y = make_classification(n_samples=1000, n_features=4,  
...                                n_informative=2, n_redundant=0,  
...                                random_state=0, shuffle=False)  
>>> clf = RandomForestClassifier(max_depth=2, random_state=0)  
>>> clf.fit(X, y)  
RandomForestClassifier(...)  
>>> print(clf.predict([[0, 0, 0, 0]]))  
[1]  
"""  
 @_deprecate_positional_args  
def __init__(self,  
             n_estimators=100, *,  
             criterion="gini",  
             max_depth=None,  
             min_samples_split=2,  
             min_samples_leaf=1,  
             min_weight_fraction_leaf=0.,  
             max_features="auto",  
             max_leaf_nodes=None,  
             min_impurity_decrease=0.,  
             min_impurity_split=None,  
             bootstrap=True,  
             oob_score=False,  
             n_jobs=None,  
             random_state=None,  
             verbose=0,  
             warm_start=False,  
             class_weight=None,  
             ccp_alpha=0.0,  
             max_samples=None):  
    super().__init__(  
        base_estimator=DecisionTreeClassifier(),  
        n_estimators=n_estimators,
```

```
        estimator_params=("criterion", "max_depth", "min_samples_split",
                           "min_samples_leaf", "min_weight_fraction_leaf",
                           "max_features", "max_leaf_nodes",
                           "min_impurity_decrease", "min_impurity_split",
                           "random_state", "ccp_alpha"),
        bootstrap=bootstrap,
        oob_score=oob_score,
        n_jobs=n_jobs,
        random_state=random_state,
        verbose=verbose,
        warm_start=warm_start,
        class_weight=class_weight,
        max_samples=max_samples)

    self.criterion = criterion
    self.max_depth = max_depth
    self.min_samples_split = min_samples_split
    self.min_samples_leaf = min_samples_leaf
    self.min_weight_fraction_leaf = min_weight_fraction_leaf
    self.max_features = max_features
    self.max_leaf_nodes = max_leaf_nodes
    self.min_impurity_decrease = min_impurity_decrease
    self.min_impurity_split = min_impurity_split
    self ccp_alpha = ccp_alpha

class RandomForestRegressor(ForestRegressor):
    """
    A random forest regressor.

    A random forest is a meta estimator that fits a number of classifying
    decision trees on various sub-samples of the dataset and uses averaging
    to improve the predictive accuracy and control over-fitting.
    The sub-sample size is controlled with the `max_samples` parameter if
    `bootstrap=True` (default), otherwise the whole dataset is used to build
    each tree.

    Read more in the :ref:`User Guide <forest>`.

    Parameters
    -----
    n_estimators : int, default=100
        The number of trees in the forest.

        .. versionchanged:: 0.22
           The default value of ``n_estimators`` changed from 10 to 100
           in 0.22.

    criterion : {"mse", "mae"}, default="mse"
        The function to measure the quality of a split. Supported criteria
        are "mse" for the mean squared error, which is equal to variance
        reduction as feature selection criterion, and "mae" for the mean
        absolute error.

        .. versionadded:: 0.18
           Mean Absolute Error (MAE) criterion.

    max_depth : int, default=None
        The maximum depth of the tree. If None, then nodes are expanded until
        all leaves are pure or until all leaves contain less than
        min_samples_split samples.

    min_samples_split : int or float, default=2
```

The minimum number of samples required to split an internal node:

- If int, then consider `min_samples_split` as the minimum number.
- If float, then `min_samples_split` is a fraction and
`ceil(min_samples_split * n_samples)` are the minimum
number of samples for each split.

.. versionchanged:: 0.18
Added float values for fractions.

min_samples_leaf : int or float, default=1

The minimum number of samples required to be at a leaf node.

A split point at any depth will only be considered if it leaves at least ``min_samples_leaf`` training samples in each of the left and right branches. This may have the effect of smoothing the model, especially in regression.

- If int, then consider `min_samples_leaf` as the minimum number.
- If float, then `min_samples_leaf` is a fraction and
`ceil(min_samples_leaf * n_samples)` are the minimum
number of samples for each node.

.. versionchanged:: 0.18
Added float values for fractions.

min_weight_fraction_leaf : float, default=0.0

The minimum weighted fraction of the sum total of weights (of all the input samples) required to be at a leaf node. Samples have equal weight when sample_weight is not provided.

max_features : {"auto", "sqrt", "log2"}, int or float, default="auto"

The number of features to consider when looking for the best split:

- If int, then consider `max_features` features at each split.
- If float, then `max_features` is a fraction and
`int(max_features * n_features)` features are considered at each split.
- If "auto", then `max_features=n_features`.
- If "sqrt", then `max_features=sqrt(n_features)`.
- If "log2", then `max_features=log2(n_features)`.
- If None, then `max_features=n_features`.

Note: the search for a split does not stop until at least one valid partition of the node samples is found, even if it requires to effectively inspect more than ``max_features`` features.

max_leaf_nodes : int, default=None

Grow trees with ``max_leaf_nodes`` in best-first fashion.

Best nodes are defined as relative reduction in impurity.

If None then unlimited number of leaf nodes.

min_impurity_decrease : float, default=0.0

A node will be split if this split induces a decrease of the impurity greater than or equal to this value.

The weighted impurity decrease equation is the following::

$$\frac{N_t}{N} \cdot (\text{impurity} - \frac{N_{t,R}}{N_t} \cdot \text{right_impurity} - \frac{N_{t,L}}{N_t} \cdot \text{left_impurity})$$

where ``N`` is the total number of samples, ``N_t`` is the number of samples at the current node, ``N_{t,L}`` is the number of samples in the left child, and ``N_{t,R}`` is the number of samples in the right child.

```N```, ```N_t```, ```N_t_R``` and ```N_t_L``` all refer to the weighted sum, if `sample_weight` is passed.

`.. versionadded:: 0.19`

`min_impurity_split : float, default=None`  
 Threshold for early stopping in tree growth. A node will split if its impurity is above the threshold, otherwise it is a leaf.

`.. deprecated:: 0.19`  
```min_impurity_split`` has been deprecated in favor of ``min_impurity_decrease`` in 0.19. The default value of ``min_impurity_split`` has changed from 1e-7 to 0 in 0.23 and it will be removed in 0.25. Use ``min_impurity_decrease`` instead.`

`bootstrap : bool, default=True`
 Whether bootstrap samples are used when building trees. If `False`, the whole dataset is used to build each tree.

`oob_score : bool, default=False`
 whether to use out-of-bag samples to estimate the R^2 on unseen data.

`n_jobs : int, default=None`
 The number of jobs to run in parallel. `:meth:`fit``, `:meth:`predict``, `:meth:`decision_path`` and `:meth:`apply`` are all parallelized over the trees. ```None``` means 1 unless in a `:obj:`joblib.parallel_backend`` context. ```-1``` means using all processors. See `:term:`Glossary <n_jobs>`` for more details.

`random_state : int or RandomState, default=None`
 Controls both the randomness of the bootstrapping of the samples used when building trees (if ```bootstrap=True```) and the sampling of the features to consider when looking for the best split at each node (if ```max_features < n_features```).
 See `:term:`Glossary <random_state>`` for details.

`verbose : int, default=0`
 Controls the verbosity when fitting and predicting.

`warm_start : bool, default=False`
 When set to ```True```, reuse the solution of the previous call to fit and add more estimators to the ensemble, otherwise, just fit a whole new forest. See `:term:`the Glossary <warm_start>``.

`ccp_alpha : non-negative float, default=0.0`
 Complexity parameter used for Minimal Cost-Complexity Pruning. The subtree with the largest cost complexity that is smaller than ```ccp_alpha``` will be chosen. By default, no pruning is performed. See `:ref:`minimal_cost_complexity_pruning`` for details.

`.. versionadded:: 0.22`

`max_samples : int or float, default=None`
 If `bootstrap` is `True`, the number of samples to draw from `X` to train each base estimator.

- If `None` (default), then draw `X.shape[0]` samples.
- If `int`, then draw `max_samples` samples.
- If `float`, then draw `max_samples * X.shape[0]` samples. Thus, `max_samples` should be in the interval `(0, 1)`.

```
.. versionadded:: 0.22

Attributes
-----
base_estimator_ : DecisionTreeRegressor
    The child estimator template used to create the collection of fitted
    sub-estimators.

estimators_ : list of DecisionTreeRegressor
    The collection of fitted sub-estimators.

feature_importances_ : ndarray of shape (n_features,)
    The impurity-based feature importances.
    The higher, the more important the feature.
    The importance of a feature is computed as the (normalized)
    total reduction of the criterion brought by that feature. It is also
    known as the Gini importance.

    Warning: impurity-based feature importances can be misleading for
    high cardinality features (many unique values). See
    :func:`sklearn.inspection.permutation_importance` as an alternative.

n_features_ : int
    The number of features when ``fit`` is performed.

n_outputs_ : int
    The number of outputs when ``fit`` is performed.

oob_score_ : float
    Score of the training dataset obtained using an out-of-bag estimate.
    This attribute exists only when ``oob_score`` is True.

oob_prediction_ : ndarray of shape (n_samples,)
    Prediction computed with out-of-bag estimate on the training set.
    This attribute exists only when ``oob_score`` is True.

See Also
-----
DecisionTreeRegressor, ExtraTreesRegressor

Notes
-----
The default values for the parameters controlling the size of the trees
(e.g. ``max_depth``, ``min_samples_leaf``, etc.) lead to fully grown and
unpruned trees which can potentially be very large on some data sets. To
reduce memory consumption, the complexity and size of the trees should be
controlled by setting those parameter values.

The features are always randomly permuted at each split. Therefore,
the best found split may vary, even with the same training data,
``max_features=n_features`` and ``bootstrap=False``, if the improvement
of the criterion is identical for several splits enumerated during the
search of the best split. To obtain a deterministic behaviour during
fitting, ``random_state`` has to be fixed.

The default value ``max_features="auto"`` uses ``n_features``
rather than ``n_features / 3``. The latter was originally suggested in
[1], whereas the former was more recently justified empirically in [2].
```

References

.. [1] L. Breiman, "Random Forests", Machine Learning, 45(1), 5-32, 2001.

```
.. [2] P. Geurts, D. Ernst., and L. Wehenkel, "Extremely randomized  
trees", Machine Learning, 63(1), 3-42, 2006.
```

Examples

```
>>> from sklearn.ensemble import RandomForestRegressor  
>>> from sklearn.datasets import make_regression  
>>> X, y = make_regression(n_features=4, n_informative=2,  
...                         random_state=0, shuffle=False)  
>>> regr = RandomForestRegressor(max_depth=2, random_state=0)  
>>> regr.fit(X, y)  
RandomForestRegressor(...)  
>>> print(regr.predict([[0, 0, 0, 0]]))  
[-8.32987858]  
"""  
 @_deprecate_positional_args  
def __init__(self,  
             n_estimators=100, *,  
             criterion="mse",  
             max_depth=None,  
             min_samples_split=2,  
             min_samples_leaf=1,  
             min_weight_fraction_leaf=0.,  
             max_features="auto",  
             max_leaf_nodes=None,  
             min_impurity_decrease=0.,  
             min_impurity_split=None,  
             bootstrap=True,  
             oob_score=False,  
             n_jobs=None,  
             random_state=None,  
             verbose=0,  
             warm_start=False,  
             ccp_alpha=0.0,  
             max_samples=None):  
    super().__init__(  
        base_estimator=DecisionTreeRegressor(),  
        n_estimators=n_estimators,  
        estimator_params=("criterion", "max_depth", "min_samples_split",  
                          "min_samples_leaf", "min_weight_fraction_leaf",  
                          "max_features", "max_leaf_nodes",  
                          "min_impurity_decrease", "min_impurity_split",  
                          "random_state", "ccp_alpha"),  
        bootstrap=bootstrap,  
        oob_score=oob_score,  
        n_jobs=n_jobs,  
        random_state=random_state,  
        verbose=verbose,  
        warm_start=warm_start,  
        max_samples=max_samples)  
  
    self.criterion = criterion  
    self.max_depth = max_depth  
    self.min_samples_split = min_samples_split  
    self.min_samples_leaf = min_samples_leaf  
    self.min_weight_fraction_leaf = min_weight_fraction_leaf  
    self.max_features = max_features  
    self.max_leaf_nodes = max_leaf_nodes  
    self.min_impurity_decrease = min_impurity_decrease  
    self.min_impurity_split = min_impurity_split  
    self ccp_alpha = ccp_alpha
```

```
class ExtraTreesClassifier(ForestClassifier):
    """
    An extra-trees classifier.

    This class implements a meta estimator that fits a number of
    randomized decision trees (a.k.a. extra-trees) on various sub-samples
    of the dataset and uses averaging to improve the predictive accuracy
    and control over-fitting.

    Read more in the :ref:`User Guide <forest>`.

    Parameters
    -----
    n_estimators : int, default=100
        The number of trees in the forest.

        .. versionchanged:: 0.22
            The default value of ``n_estimators`` changed from 10 to 100
            in 0.22.

    criterion : {"gini", "entropy"}, default="gini"
        The function to measure the quality of a split. Supported criteria are
        "gini" for the Gini impurity and "entropy" for the information gain.

    max_depth : int, default=None
        The maximum depth of the tree. If None, then nodes are expanded until
        all leaves are pure or until all leaves contain less than
        min_samples_split samples.

    min_samples_split : int or float, default=2
        The minimum number of samples required to split an internal node:

        - If int, then consider `min_samples_split` as the minimum number.
        - If float, then `min_samples_split` is a fraction and
          `ceil(min_samples_split * n_samples)` are the minimum
          number of samples for each split.

        .. versionchanged:: 0.18
            Added float values for fractions.

    min_samples_leaf : int or float, default=1
        The minimum number of samples required to be at a leaf node.
        A split point at any depth will only be considered if it leaves at
        least ``min_samples_leaf`` training samples in each of the left and
        right branches. This may have the effect of smoothing the model,
        especially in regression.

        - If int, then consider `min_samples_leaf` as the minimum number.
        - If float, then `min_samples_leaf` is a fraction and
          `ceil(min_samples_leaf * n_samples)` are the minimum
          number of samples for each node.

        .. versionchanged:: 0.18
            Added float values for fractions.

    min_weight_fraction_leaf : float, default=0.0
        The minimum weighted fraction of the sum total of weights (of all
        the input samples) required to be at a leaf node. Samples have
        equal weight when sample_weight is not provided.

    max_features : {"auto", "sqrt", "log2"}, int or float, default="auto"
        The number of features to consider when looking for the best split:
```

```
- If int, then consider `max_features` features at each split.  
- If float, then `max_features` is a fraction and  
`int(max_features * n_features)` features are considered at each  
split.  
- If "auto", then `max_features=sqrt(n_features)`.  
- If "sqrt", then `max_features=sqrt(n_features)`.  
- If "log2", then `max_features=log2(n_features)`.  
- If None, then `max_features=n_features`.
```

Note: the search for a split does not stop until at least one valid partition of the node samples is found, even if it requires to effectively inspect more than ``max_features`` features.

```
max_leaf_nodes : int, default=None  
Grow trees with ``max_leaf_nodes`` in best-first fashion.  
Best nodes are defined as relative reduction in impurity.  
If None then unlimited number of leaf nodes.
```

```
min_impurity_decrease : float, default=0.0  
A node will be split if this split induces a decrease of the impurity  
greater than or equal to this value.
```

The weighted impurity decrease equation is the following::

$$\frac{N_t}{N} \cdot (\text{impurity} - \frac{N_{t,R}}{N_t} \cdot \text{right_impurity} - \frac{N_{t,L}}{N_t} \cdot \text{left_impurity})$$

where ``N`` is the total number of samples, ``N_t`` is the number of samples at the current node, ``N_{t,L}`` is the number of samples in the left child, and ``N_{t,R}`` is the number of samples in the right child.

``N``, ``N_t``, ``N_{t,R}`` and ``N_{t,L}`` all refer to the weighted sum, if ``sample_weight`` is passed.

.. versionadded:: 0.19

```
min_impurity_split : float, default=None  
Threshold for early stopping in tree growth. A node will split  
if its impurity is above the threshold, otherwise it is a leaf.
```

.. deprecated:: 0.19
``min_impurity_split`` has been deprecated in favor of
``min_impurity_decrease`` in 0.19. The default value of
``min_impurity_split`` has changed from 1e-7 to 0 in 0.23 and it
will be removed in 0.25. Use ``min_impurity_decrease`` instead.

```
bootstrap : bool, default=False  
Whether bootstrap samples are used when building trees. If False, the  
whole dataset is used to build each tree.
```

```
oob_score : bool, default=False  
Whether to use out-of-bag samples to estimate  
the generalization accuracy.
```

```
n_jobs : int, default=None  
The number of jobs to run in parallel. :meth:`fit`, :meth:`predict`,  
:meth:`decision_path` and :meth:`apply` are all parallelized over the  
trees. ``None`` means 1 unless in a :obj:`joblib.parallel_backend`  
context. ``-1`` means using all processors. See :term:`Glossary`  
<n_jobs> for more details.
```

```
random_state : int, RandomState, default=None  
Controls 3 sources of randomness:
```

- the bootstrapping of the samples used when building trees (if ``bootstrap=True``)
- the sampling of the features to consider when looking for the best split at each node (if ``max_features < n_features``)
- the draw of the splits for each of the `max_features`

See :term:`Glossary <random_state>` for details.

`verbose : int, default=0`

Controls the verbosity when fitting and predicting.

`warm_start : bool, default=False`

When set to ``True``, reuse the solution of the previous call to fit and add more estimators to the ensemble, otherwise, just fit a whole new forest. See :term:`the Glossary <warm_start>`.

`class_weight : {"balanced", "balanced_subsample"}, dict or list of dicts, \ default=None`

Weights associated with classes in the form ``{class_label: weight}``.

If not given, all classes are supposed to have weight one. For multi-output problems, a list of dicts can be provided in the same order as the columns of `y`.

Note that for multioutput (including multilabel) weights should be defined for each class of every column in its own dict. For example, for four-class multilabel classification weights should be

`[{0: 1, 1: 1}, {0: 1, 1: 5}, {0: 1, 1: 1}, {0: 1, 1: 1}]` instead of `[{1:1}, {2:5}, {3:1}, {4:1}]`.

The "balanced" mode uses the values of `y` to automatically adjust weights inversely proportional to class frequencies in the input data as ```n_samples / (n_classes * np.bincount(y))```

The "balanced_subsample" mode is the same as "balanced" except that weights are computed based on the bootstrap sample for every tree grown.

For multi-output, the weights of each column of `y` will be multiplied.

Note that these weights will be multiplied with `sample_weight` (passed through the `fit` method) if `sample_weight` is specified.

`ccp_alpha : non-negative float, default=0.0`

Complexity parameter used for Minimal Cost-Complexity Pruning. The subtree with the largest cost complexity that is smaller than ```ccp_alpha``` will be chosen. By default, no pruning is performed. See :ref:`minimal_cost_complexity_pruning` for details.

.. versionadded:: 0.22

`max_samples : int or float, default=None`

If `bootstrap` is `True`, the number of samples to draw from `X` to train each base estimator.

- If `None` (default), then draw `X.shape[0]` samples.

- If `int`, then draw `max_samples` samples.

- If `float`, then draw `max_samples * X.shape[0]` samples. Thus, `max_samples` should be in the interval `(0, 1)`.

.. versionadded:: 0.22

Attributes

`base_estimator_` : ExtraTreesClassifier
The child estimator template used to create the collection of fitted sub-estimators.

`estimators_` : list of DecisionTreeClassifier
The collection of fitted sub-estimators.

`classes_` : ndarray of shape (n_classes,) or a list of such arrays
The classes labels (single output problem), or a list of arrays of class labels (multi-output problem).

`n_classes_` : int or list
The number of classes (single output problem), or a list containing the number of classes for each output (multi-output problem).

`feature_importances_` : ndarray of shape (n_features,)
The impurity-based feature importances.
The higher, the more important the feature.
The importance of a feature is computed as the (normalized) total reduction of the criterion brought by that feature. It is also known as the Gini importance.

Warning: impurity-based feature importances can be misleading for high cardinality features (many unique values). See :func:`sklearn.inspection.permutation_importance` as an alternative.

`n_features_` : int
The number of features when ``fit`` is performed.

`n_outputs_` : int
The number of outputs when ``fit`` is performed.

`oob_score_` : float
Score of the training dataset obtained using an out-of-bag estimate.
This attribute exists only when ``oob_score`` is True.

`oob_decision_function_` : ndarray of shape (n_samples, n_classes)
Decision function computed with out-of-bag estimate on the training set. If n_estimators is small it might be possible that a data point was never left out during the bootstrap. In this case, `oob_decision_function` might contain NaN. This attribute exists only when ``oob_score`` is True.

See Also

`sklearn.tree.ExtraTreeClassifier` : Base classifier for this ensemble.
`RandomForestClassifier` : Ensemble Classifier based on trees with optimal splits.

Notes

The default values for the parameters controlling the size of the trees (e.g. ``max_depth``, ``min_samples_leaf``, etc.) lead to fully grown and unpruned trees which can potentially be very large on some data sets. To reduce memory consumption, the complexity and size of the trees should be controlled by setting those parameter values.

References

.. [1] P. Geurts, D. Ernst., and L. Wehenkel, "Extremely randomized trees", Machine Learning, 63(1), 3-42, 2006.

Examples

```
-----
>>> from sklearn.ensemble import ExtraTreesClassifier
>>> from sklearn.datasets import make_classification
>>> X, y = make_classification(n_features=4, random_state=0)
>>> clf = ExtraTreesClassifier(n_estimators=100, random_state=0)
>>> clf.fit(X, y)
ExtraTreesClassifier(random_state=0)
>>> clf.predict([[0, 0, 0, 0]])
array([1])
"""
@_deprecate_positional_args
def __init__(self,
             n_estimators=100, *,
             criterion="gini",
             max_depth=None,
             min_samples_split=2,
             min_samples_leaf=1,
             min_weight_fraction_leaf=0.,
             max_features="auto",
             max_leaf_nodes=None,
             min_impurity_decrease=0.,
             min_impurity_split=None,
             bootstrap=False,
             oob_score=False,
             n_jobs=None,
             random_state=None,
             verbose=0,
             warm_start=False,
             class_weight=None,
             ccp_alpha=0.0,
             max_samples=None):
    super().__init__(
        base_estimator=ExtraTreeClassifier(),
        n_estimators=n_estimators,
        estimator_params=("criterion", "max_depth", "min_samples_split",
                          "min_samples_leaf", "min_weight_fraction_leaf",
                          "max_features", "max_leaf_nodes",
                          "min_impurity_decrease", "min_impurity_split",
                          "random_state", "ccp_alpha"),
        bootstrap=bootstrap,
        oob_score=oob_score,
        n_jobs=n_jobs,
        random_state=random_state,
        verbose=verbose,
        warm_start=warm_start,
        class_weight=class_weight,
        max_samples=max_samples)

    self.criterion = criterion
    self.max_depth = max_depth
    self.min_samples_split = min_samples_split
    self.min_samples_leaf = min_samples_leaf
    self.min_weight_fraction_leaf = min_weight_fraction_leaf
    self.max_features = max_features
    self.max_leaf_nodes = max_leaf_nodes
    self.min_impurity_decrease = min_impurity_decrease
    self.min_impurity_split = min_impurity_split
    self ccp_alpha = ccp_alpha

class ExtraTreesRegressor(ForestRegressor):
    """
```

An extra-trees regressor.

This class implements a meta estimator that fits a number of randomized decision trees (a.k.a. extra-trees) on various sub-samples of the dataset and uses averaging to improve the predictive accuracy and control over-fitting.

Read more in the :ref:`User Guide <forest>`.

Parameters

`n_estimators` : int, default=100
The number of trees in the forest.

.. versionchanged:: 0.22
The default value of ```n_estimators``` changed from 10 to 100 in 0.22.

`criterion` : {"mse", "mae"}, default="mse"
The function to measure the quality of a split. Supported criteria are "mse" for the mean squared error, which is equal to variance reduction as feature selection criterion, and "mae" for the mean absolute error.

.. versionadded:: 0.18
Mean Absolute Error (MAE) criterion.

`max_depth` : int, default=None
The maximum depth of the tree. If None, then nodes are expanded until all leaves are pure or until all leaves contain less than `min_samples_split` samples.

`min_samples_split` : int or float, default=2
The minimum number of samples required to split an internal node:

- If int, then consider `'min_samples_split'` as the minimum number.
- If float, then `'min_samples_split'` is a fraction and `'ceil(min_samples_split * n_samples)'` are the minimum number of samples for each split.

.. versionchanged:: 0.18
Added float values for fractions.

`min_samples_leaf` : int or float, default=1
The minimum number of samples required to be at a leaf node.
A split point at any depth will only be considered if it leaves at least ```min_samples_leaf``` training samples in each of the left and right branches. This may have the effect of smoothing the model, especially in regression.

- If int, then consider `'min_samples_leaf'` as the minimum number.
- If float, then `'min_samples_leaf'` is a fraction and `'ceil(min_samples_leaf * n_samples)'` are the minimum number of samples for each node.

.. versionchanged:: 0.18
Added float values for fractions.

`min_weight_fraction_leaf` : float, default=0.0
The minimum weighted fraction of the sum total of weights (of all the input samples) required to be at a leaf node. Samples have equal weight when `sample_weight` is not provided.

```
max_features : {"auto", "sqrt", "log2"} int or float, default="auto"
    The number of features to consider when looking for the best split:
```

- If int, then consider `max_features` features at each split.
- If float, then `max_features` is a fraction and
`int(max_features * n_features)` features are considered at each split.
- If "auto", then `max_features=n_features`.
- If "sqrt", then `max_features=sqrt(n_features)`.
- If "log2", then `max_features=log2(n_features)`.
- If None, then `max_features=n_features`.

Note: the search for a split does not stop until at least one valid partition of the node samples is found, even if it requires to effectively inspect more than ``max_features`` features.

```
max_leaf_nodes : int, default=None
    Grow trees with ``max_leaf_nodes`` in best-first fashion.
    Best nodes are defined as relative reduction in impurity.
    If None then unlimited number of leaf nodes.
```

```
min_impurity_decrease : float, default=0.0
    A node will be split if this split induces a decrease of the impurity
    greater than or equal to this value.
```

The weighted impurity decrease equation is the following::

$$\frac{N_t}{N} \cdot (\text{impurity} - \frac{N_{t,R}}{N_t} \cdot \text{right_impurity} - \frac{N_{t,L}}{N_t} \cdot \text{left_impurity})$$

where ``N`` is the total number of samples, ``N_t`` is the number of samples at the current node, ``N_{t,L}`` is the number of samples in the left child, and ``N_{t,R}`` is the number of samples in the right child.

``N``, ``N_t``, ``N_{t,R}`` and ``N_{t,L}`` all refer to the weighted sum, if ``sample_weight`` is passed.

.. versionadded:: 0.19

```
min_impurity_split : float, default=None
    Threshold for early stopping in tree growth. A node will split
    if its impurity is above the threshold, otherwise it is a leaf.
```

.. deprecated:: 0.19
``min_impurity_split`` has been deprecated in favor of
``min_impurity_decrease`` in 0.19. The default value of
``min_impurity_split`` has changed from 1e-7 to 0 in 0.23 and it
will be removed in 0.25. Use ``min_impurity_decrease`` instead.

```
bootstrap : bool, default=False
    Whether bootstrap samples are used when building trees. If False, the
    whole dataset is used to build each tree.
```

```
oob_score : bool, default=False
    Whether to use out-of-bag samples to estimate the R^2 on unseen data.
```

```
n_jobs : int, default=None
    The number of jobs to run in parallel. :meth:`fit`, :meth:`predict`,
    :meth:`decision_path` and :meth:`apply` are all parallelized over the
    trees. ``None`` means 1 unless in a :obj:`joblib.parallel_backend`
    context. ``-1`` means using all processors. See :term:`Glossary
    <n_jobs>` for more details.
```

```
random_state : int or RandomState, default=None
    Controls 3 sources of randomness:
        - the bootstrapping of the samples used when building trees
          (if ``bootstrap=True``)
        - the sampling of the features to consider when looking for the best
          split at each node (if ``max_features < n_features``)
        - the draw of the splits for each of the `max_features`

    See :term:`Glossary <random_state>` for details.

verbose : int, default=0
    Controls the verbosity when fitting and predicting.

warm_start : bool, default=False
    When set to ``True``, reuse the solution of the previous call to fit
    and add more estimators to the ensemble, otherwise, just fit a whole
    new forest. See :term:`the Glossary <warm_start>`.

ccp_alpha : non-negative float, default=0.0
    Complexity parameter used for Minimal Cost-Complexity Pruning. The
    subtree with the largest cost complexity that is smaller than
    ``ccp_alpha`` will be chosen. By default, no pruning is performed. See
    :ref:`minimal_cost_complexity_pruning` for details.

.. versionadded:: 0.22

max_samples : int or float, default=None
    If bootstrap is True, the number of samples to draw from X
    to train each base estimator.

    - If None (default), then draw `X.shape[0]` samples.
    - If int, then draw `max_samples` samples.
    - If float, then draw `max_samples * X.shape[0]` samples. Thus,
      `max_samples` should be in the interval `(0, 1)`.

.. versionadded:: 0.22

Attributes
-----
base_estimator_ : ExtraTreeRegressor
    The child estimator template used to create the collection of fitted
    sub-estimators.

estimators_ : list of DecisionTreeRegressor
    The collection of fitted sub-estimators.

feature_importances_ : ndarray of shape (n_features,)
    The impurity-based feature importances.
    The higher, the more important the feature.
    The importance of a feature is computed as the (normalized)
    total reduction of the criterion brought by that feature. It is also
    known as the Gini importance.

    Warning: impurity-based feature importances can be misleading for
    high cardinality features (many unique values). See
    :func:`sklearn.inspection.permutation_importance` as an alternative.

n_features_ : int
    The number of features.

n_outputs_ : int
    The number of outputs.
```

```
oob_score_ : float
    Score of the training dataset obtained using an out-of-bag estimate.
    This attribute exists only when ``oob_score`` is True.

oob_prediction_ : ndarray of shape (n_samples,)
    Prediction computed with out-of-bag estimate on the training set.
    This attribute exists only when ``oob_score`` is True.

See Also
-----
sklearn.tree.ExtraTreeRegressor: Base estimator for this ensemble.
RandomForestRegressor: Ensemble regressor using trees with optimal splits.

Notes
-----
The default values for the parameters controlling the size of the trees
(e.g. ``max_depth``, ``min_samples_leaf``, etc.) lead to fully grown and
unpruned trees which can potentially be very large on some data sets. To
reduce memory consumption, the complexity and size of the trees should be
controlled by setting those parameter values.

References
-----
.. [1] P. Geurts, D. Ernst., and L. Wehenkel, "Extremely randomized trees",
Machine Learning, 63(1), 3-42, 2006.

Examples
-----
>>> from sklearn.datasets import load_diabetes
>>> from sklearn.model_selection import train_test_split
>>> from sklearn.ensemble import ExtraTreesRegressor
>>> X, y = load_diabetes(return_X_y=True)
>>> X_train, X_test, y_train, y_test = train_test_split(
...     X, y, random_state=0)
>>> reg = ExtraTreesRegressor(n_estimators=100, random_state=0).fit(
...     X_train, y_train)
>>> reg.score(X_test, y_test)
0.2708...
"""
 @_deprecate_positional_args
def __init__(self,
             n_estimators=100, *,
             criterion="mse",
             max_depth=None,
             min_samples_split=2,
             min_samples_leaf=1,
             min_weight_fraction_leaf=0.,
             max_features="auto",
             max_leaf_nodes=None,
             min_impurity_decrease=0.,
             min_impurity_split=None,
             bootstrap=False,
             oob_score=False,
             n_jobs=None,
             random_state=None,
             verbose=0,
             warm_start=False,
             ccp_alpha=0.0,
             max_samples=None):
    super().__init__(
        base_estimator=ExtraTreeRegressor(),
        n_estimators=n_estimators,
```

```

estimator_params=("criterion", "max_depth", "min_samples_split",
                  "min_samples_leaf", "min_weight_fraction_leaf",
                  "max_features", "max_leaf_nodes",
                  "min_impurity_decrease", "min_impurity_split",
                  "random_state", "ccp_alpha"),
bootstrap=bootstrap,
oob_score=oob_score,
n_jobs=n_jobs,
random_state=random_state,
verbose=verbose,
warm_start=warm_start,
max_samples=max_samples)

self.criterion = criterion
self.max_depth = max_depth
self.min_samples_split = min_samples_split
self.min_samples_leaf = min_samples_leaf
self.min_weight_fraction_leaf = min_weight_fraction_leaf
self.max_features = max_features
self.max_leaf_nodes = max_leaf_nodes
self.min_impurity_decrease = min_impurity_decrease
self.min_impurity_split = min_impurity_split
self ccp_alpha = ccp_alpha

"""

class RandomTreesEmbedding(BaseForest):
    """
    An ensemble of totally random trees.

    An unsupervised transformation of a dataset to a high-dimensional sparse representation. A datapoint is coded according to which leaf of each tree it is sorted into. Using a one-hot encoding of the leaves, this leads to a binary coding with as many ones as there are trees in the forest.

    The dimensionality of the resulting representation is ``n_out <= n_estimators * max_leaf_nodes``. If ``max_leaf_nodes == None``, the number of leaf nodes is at most ``n_estimators * 2 ** max_depth``.

    Read more in the :ref:`User Guide <random_trees_embedding>`.

    Parameters
    -----
    n_estimators : int, default=100
        Number of trees in the forest.

        .. versionchanged:: 0.22
            The default value of ``n_estimators`` changed from 10 to 100 in 0.22.

    max_depth : int, default=5
        The maximum depth of each tree. If None, then nodes are expanded until all leaves are pure or until all leaves contain less than min_samples_split samples.

    min_samples_split : int or float, default=2
        The minimum number of samples required to split an internal node:
        - If int, then consider `min_samples_split` as the minimum number.
        - If float, then `min_samples_split` is a fraction and `ceil(min_samples_split * n_samples)` is the minimum number of samples for each split.
    
```

```

.. versionchanged:: 0.18
    Added float values for fractions.

min_samples_leaf : int or float, default=1
The minimum number of samples required to be at a leaf node.
A split point at any depth will only be considered if it leaves at
least ``min_samples_leaf`` training samples in each of the left and
right branches. This may have the effect of smoothing the model,
especially in regression.

- If int, then consider `min_samples_leaf` as the minimum number.
- If float, then `min_samples_leaf` is a fraction and
`ceil(min_samples_leaf * n_samples)` is the minimum
number of samples for each node.

.. versionchanged:: 0.18
    Added float values for fractions.

min_weight_fraction_leaf : float, default=0.0
The minimum weighted fraction of the sum total of weights (of all
the input samples) required to be at a leaf node. Samples have
equal weight when sample_weight is not provided.

max_leaf_nodes : int, default=None
Grow trees with ``max_leaf_nodes`` in best-first fashion.
Best nodes are defined as relative reduction in impurity.
If None then unlimited number of leaf nodes.

min_impurity_decrease : float, default=0.0
A node will be split if this split induces a decrease of the impurity
greater than or equal to this value.

The weighted impurity decrease equation is the following::

    N_t / N * (impurity - N_t_R / N_t * right_impurity
                - N_t_L / N_t * left_impurity)

where ``N`` is the total number of samples, ``N_t`` is the number of
samples at the current node, ``N_t_L`` is the number of samples in the
left child, and ``N_t_R`` is the number of samples in the right child.

``N``, ``N_t``, ``N_t_R`` and ``N_t_L`` all refer to the weighted sum,
if ``sample_weight`` is passed.

.. versionadded:: 0.19

min_impurity_split : float, default=None
Threshold for early stopping in tree growth. A node will split
if its impurity is above the threshold, otherwise it is a leaf.

.. deprecated:: 0.19
    ``min_impurity_split`` has been deprecated in favor of
    ``min_impurity_decrease`` in 0.19. The default value of
    ``min_impurity_split`` has changed from 1e-7 to 0 in 0.23 and it
    will be removed in 0.25. Use ``min_impurity_decrease`` instead.

sparse_output : bool, default=True
Whether or not to return a sparse CSR matrix, as default behavior,
or to return a dense array compatible with dense pipeline operators.

n_jobs : int, default=None
The number of jobs to run in parallel. :meth:`fit`, :meth:`transform`,
:meth:`decision_path` and :meth:`apply` are all parallelized over the

```

trees. ``None`` means 1 unless in a :obj:`joblib.parallel_backend` context. ``-1`` means using all processors. See :term:`Glossary <n_jobs>` for more details.

`random_state` : int or RandomState, default=None
Controls the generation of the random `y` used to fit the trees and the draw of the splits for each feature at the trees' nodes. See :term:`Glossary <random_state>` for details.

`verbose` : int, default=0
Controls the verbosity when fitting and predicting.

`warm_start` : bool, default=False
When set to ``True``, reuse the solution of the previous call to fit and add more estimators to the ensemble, otherwise, just fit a whole new forest. See :term:`the Glossary <warm_start>`.

Attributes

`estimators_` : list of DecisionTreeClassifier
The collection of fitted sub-estimators.

References

.. [1] P. Geurts, D. Ernst., and L. Wehenkel, "Extremely randomized trees", Machine Learning, 63(1), 3-42, 2006.
.. [2] Moosmann, F. and Triggs, B. and Jurie, F. "Fast discriminative visual codebooks using randomized clustering forests"
NIPS 2007

Examples

`>>> from sklearn.ensemble import RandomTreesEmbedding`
`>>> X = [[0,0], [1,0], [0,1], [-1,0], [0,-1]]`
`>>> random_trees = RandomTreesEmbedding(
... n_estimators=5, random_state=0, max_depth=1).fit(X)`
`>>> X_sparse_embedding = random_trees.transform(X)`
`>>> X_sparse_embedding.toarray()`
`array([[0., 1., 1., 0., 1., 0., 0., 1., 1., 0.],`
 `[0., 1., 1., 0., 1., 0., 0., 1., 1., 0.],`
 `[0., 1., 0., 1., 0., 1., 0., 1., 0., 1.],`
 `[1., 0., 1., 0., 1., 0., 1., 0., 1., 0.],`
 `[0., 1., 1., 0., 1., 0., 0., 1., 1., 0.]])`
`....`
`criterion = 'mse'`
`max_features = 1`
`@_deprecate_positional_args`
`def __init__(self,`
 `n_estimators=100, *,`
 `max_depth=5,`
 `min_samples_split=2,`
 `min_samples_leaf=1,`
 `min_weight_fraction_leaf=0.,`
 `max_leaf_nodes=None,`
 `min_impurity_decrease=0.,`
 `min_impurity_split=None,`
 `sparse_output=True,`
 `n_jobs=None,`
 `random_state=None,`
 `verbose=0,`
 `warm_start=False):`

```

super().__init__(
    base_estimator=ExtraTreeRegressor(),
    n_estimators=n_estimators,
    estimator_params=("criterion", "max_depth", "min_samples_split",
                      "min_samples_leaf", "min_weight_fraction_leaf",
                      "max_features", "max_leaf_nodes",
                      "min_impurity_decrease", "min_impurity_split",
                      "random_state"),
    bootstrap=False,
    oob_score=False,
    n_jobs=n_jobs,
    random_state=random_state,
    verbose=verbose,
    warm_start=warm_start,
    max_samples=None)

self.max_depth = max_depth
self.min_samples_split = min_samples_split
self.min_samples_leaf = min_samples_leaf
self.min_weight_fraction_leaf = min_weight_fraction_leaf
self.max_leaf_nodes = max_leaf_nodes
self.min_impurity_decrease = min_impurity_decrease
self.min_impurity_split = min_impurity_split
self.sparse_output = sparse_output

def _set_oob_score(self, X, y):
    raise NotImplementedError("OOB score not supported by tree embedding")

def fit(self, X, y=None, sample_weight=None):
    """
    Fit estimator.

    Parameters
    -----
    X : {array-like, sparse matrix} of shape (n_samples, n_features)
        The input samples. Use ``dtype=np.float32`` for maximum
        efficiency. Sparse matrices are also supported, use sparse
        ``csc_matrix`` for maximum efficiency.

    y : Ignored
        Not used, present for API consistency by convention.

    sample_weight : array-like of shape (n_samples,), default=None
        Sample weights. If None, then samples are equally weighted. Splits
        that would create child nodes with net zero or negative weight are
        ignored while searching for a split in each node. In the case of
        classification, splits are also ignored if they would result in any
        single class carrying a negative weight in either child node.

    Returns
    -----
    self : object
    """
    self.fit_transform(X, y, sample_weight=sample_weight)
    return self

def fit_transform(self, X, y=None, sample_weight=None):
    """
    Fit estimator and transform dataset.

    Parameters
    -----

```

```

X : {array-like, sparse matrix} of shape (n_samples, n_features)
    Input data used to build forests. Use ``dtype=np.float32`` for
    maximum efficiency.

y : Ignored
    Not used, present for API consistency by convention.

sample_weight : array-like of shape (n_samples,), default=None
    Sample weights. If None, then samples are equally weighted. Splits
    that would create child nodes with net zero or negative weight are
    ignored while searching for a split in each node. In the case of
    classification, splits are also ignored if they would result in any
    single class carrying a negative weight in either child node.

Returns
-----
X_transformed : sparse matrix of shape (n_samples, n_out)
    Transformed dataset.

"""
X = check_array(X, accept_sparse=['csc'])
if issparse(X):
    # Pre-sort indices to avoid that each individual tree of the
    # ensemble sorts the indices.
    X.sort_indices()

rnd = check_random_state(self.random_state)
y = rnd.uniform(size=X.shape[0])
super().fit(X, y, sample_weight=sample_weight)

self.one_hot_encoder_ = OneHotEncoder(sparse=self.sparse_output)
return self.one_hot_encoder_.fit_transform(self.apply(X))

def transform(self, X):
    """
    Transform dataset.

Parameters
-----
X : {array-like, sparse matrix} of shape (n_samples, n_features)
    Input data to be transformed. Use ``dtype=np.float32`` for maximum
    efficiency. Sparse matrices are also supported, use sparse
    ``csr_matrix`` for maximum efficiency.

Returns
-----
X_transformed : sparse matrix of shape (n_samples, n_out)
    Transformed dataset.

"""
check_is_fitted(self)
return self.one_hot_encoder_.transform(self.apply(X))

```

https://github.com/scikit-learn/scikit-learn/blob/master/sklearn/naive_bayes.py

```
# -*- coding: utf-8 -*-
"""
The :mod:`sklearn.naive_bayes` module implements Naive Bayes algorithms. These
are supervised learning methods based on applying Bayes' theorem with strong
(naive) feature independence assumptions.
"""

# Author: Vincent Michel <vincent.michel@inria.fr>
#         Minor fixes by Fabian Pedregosa
#         Amit Aides <amitibo@tx.technion.ac.il>
#         Yehuda Finkelstein <yehudaf@tx.technion.ac.il>
#         Lars Buitinck
#         Jan Hendrik Metzen <jhm@informatik.uni-bremen.de>
#         (parts based on earlier work by Mathieu Blondel)
#
# License: BSD 3 clause
import warnings

from abc import ABCMeta, abstractmethod

import numpy as np
from scipy.special import logsumexp

from .base import BaseEstimator, ClassifierMixin
from .preprocessing import binarize
from .preprocessing import LabelBinarizer
from .preprocessing import label_binarize
from .utils import check_X_y, check_array, deprecated
from .utils.extmath import safe_sparse_dot
from .utils.multiclass import _check_partial_fit_first_call
from .utils.validation import check_is_fitted, check_non_negative, column_or_1d
from .utils.validation import _check_sample_weight
from .utils.validation import _deprecate_positional_args

__all__ = ['BernoulliNB', 'GaussianNB', 'MultinomialNB', 'ComplementNB',
          'CategoricalNB']

class _BaseNB(ClassifierMixin, BaseEstimator, metaclass=ABCMeta):
    """Abstract base class for naive Bayes estimators"""

    @abstractmethod
    def _joint_log_likelihood(self, X):
        """Compute the unnormalized posterior log probability of X

        I.e. ``log P(c) + log P(x|c)`` for all rows x of X, as an array-like of
        shape (n_classes, n_samples).

        Input is passed to _joint_log_likelihood as-is by predict,
        predict_proba and predict_log_proba.
        """

    def _check_X(self, X):
        """To be overridden in subclasses with the actual checks."""
        # Note that this is not marked @abstractmethod as long as the
```

```

# deprecated public alias sklearn.naive_bayes.BayesNB exists
# (until 0.24) to preserve backward compat for 3rd party projects
# with existing derived classes.
return X

def predict(self, X):
    """
    Perform classification on an array of test vectors X.

    Parameters
    -----
    X : array-like of shape (n_samples, n_features)

    Returns
    -----
    C : ndarray of shape (n_samples,)
        Predicted target values for X
    """
    check_is_fitted(self)
    X = self._check_X(X)
    jll = self._joint_log_likelihood(X)
    return self.classes_[np.argmax(jll, axis=1)]

def predict_log_proba(self, X):
    """
    Return log-probability estimates for the test vector X.

    Parameters
    -----
    X : array-like of shape (n_samples, n_features)

    Returns
    -----
    C : array-like of shape (n_samples, n_classes)
        Returns the log-probability of the samples for each class in
        the model. The columns correspond to the classes in sorted
        order, as they appear in the attribute :term:`classes_`.
    """
    check_is_fitted(self)
    X = self._check_X(X)
    jll = self._joint_log_likelihood(X)
    # normalize by  $P(x) = P(f_1, \dots, f_n)$ 
    log_prob_x = logsumexp(jll, axis=1)
    return jll - np.atleast_2d(log_prob_x).T

def predict_proba(self, X):
    """
    Return probability estimates for the test vector X.

    Parameters
    -----
    X : array-like of shape (n_samples, n_features)

    Returns
    -----
    C : array-like of shape (n_samples, n_classes)
        Returns the probability of the samples for each class in
        the model. The columns correspond to the classes in sorted
        order, as they appear in the attribute :term:`classes_`.
    """
    return np.exp(self.predict_log_proba(X))

```

```
class GaussianNB(_BaseNB):
    """
    Gaussian Naive Bayes (GaussianNB)

    Can perform online updates to model parameters via :meth:`partial_fit`.
    For details on algorithm used to update feature means and variance online,
    see Stanford CS tech report STAN-CS-79-773 by Chan, Golub, and LeVeque:

    http://i.stanford.edu/pub/cstr/reports/cs/tr/79/773/CS-TR-79-773.pdf

    Read more in the :ref:`User Guide <gaussian_naive_bayes>`.

    Parameters
    -----
    priors : array-like of shape (n_classes,)
        Prior probabilities of the classes. If specified the priors are not
        adjusted according to the data.

    var_smoothing : float, default=1e-9
        Portion of the largest variance of all features that is added to
        variances for calculation stability.

    Attributes
    -----
    class_count_ : ndarray of shape (n_classes,)
        number of training samples observed in each class.

    class_prior_ : ndarray of shape (n_classes,)
        probability of each class.

    classes_ : ndarray of shape (n_classes,)
        class labels known to the classifier

    epsilon_ : float
        absolute additive value to variances

    sigma_ : ndarray of shape (n_classes, n_features)
        variance of each feature per class

    theta_ : ndarray of shape (n_classes, n_features)
        mean of each feature per class

    Examples
    -----
    >>> import numpy as np
    >>> X = np.array([[-1, -1], [-2, -1], [-3, -2], [1, 1], [2, 1], [3, 2]])
    >>> Y = np.array([1, 1, 1, 2, 2, 2])
    >>> from sklearn.naive_bayes import GaussianNB
    >>> clf = GaussianNB()
    >>> clf.fit(X, Y)
    GaussianNB()
    >>> print(clf.predict([-0.8, -1]))
    [1]
    >>> clf_pf = GaussianNB()
    >>> clf_pf.partial_fit(X, Y, np.unique(Y))
    GaussianNB()
    >>> print(clf_pf.predict([-0.8, -1]))
    [1]
    """
    @_deprecated_positional_args
    def __init__(self, *, priors=None, var_smoothing=1e-9):
        self.priors = priors
```

```

    self.var_smoothing = var_smoothing

def fit(self, X, y, sample_weight=None):
    """Fit Gaussian Naive Bayes according to X, y

    Parameters
    -----
    X : array-like of shape (n_samples, n_features)
        Training vectors, where n_samples is the number of samples
        and n_features is the number of features.

    y : array-like of shape (n_samples,)
        Target values.

    sample_weight : array-like of shape (n_samples,), default=None
        Weights applied to individual samples (1. for unweighted).

        .. versionadded:: 0.17
            Gaussian Naive Bayes supports fitting with *sample_weight*.

    Returns
    -----
    self : object
    """
    X, y = self._validate_data(X, y)
    y = column_or_1d(y, warn=True)
    return self._partial_fit(X, y, np.unique(y), _refit=True,
                           sample_weight=sample_weight)

def _check_X(self, X):
    return check_array(X)

@staticmethod
def _update_mean_variance(n_past, mu, var, X, sample_weight=None):
    """Compute online update of Gaussian mean and variance.

    Given starting sample count, mean, and variance, a new set of
    points X, and optionally sample weights, return the updated mean and
    variance. (NB - each dimension (column) in X is treated as independent
    -- you get variance, not covariance).

    Can take scalar mean and variance, or vector mean and variance to
    simultaneously update a number of independent Gaussians.

    See Stanford CS tech report STAN-CS-79-773 by Chan, Golub, and LeVeque:
    http://i.stanford.edu/pub/cstr/reports/cs/tr/79/773/CS-TR-79-773.pdf

    Parameters
    -----
    n_past : int
        Number of samples represented in old mean and variance. If sample
        weights were given, this should contain the sum of sample
        weights represented in old mean and variance.

    mu : array-like of shape (number of Gaussians,)
        Means for Gaussians in original set.

    var : array-like of shape (number of Gaussians,)
        Variances for Gaussians in original set.

    sample_weight : array-like of shape (n_samples,), default=None
        Weights applied to individual samples (1. for unweighted).
    """

```

```

>Returns
-----
total_mu : array-like of shape (number of Gaussians,)
    Updated mean for each Gaussian over the combined set.

total_var : array-like of shape (number of Gaussians,)
    Updated variance for each Gaussian over the combined set.
"""
if X.shape[0] == 0:
    return mu, var

# Compute (potentially weighted) mean and variance of new datapoints
if sample_weight is not None:
    n_new = float(sample_weight.sum())
    new_mu = np.average(X, axis=0, weights=sample_weight)
    new_var = np.average((X - new_mu) ** 2, axis=0,
                         weights=sample_weight)
else:
    n_new = X.shape[0]
    new_var = np.var(X, axis=0)
    new_mu = np.mean(X, axis=0)

if n_past == 0:
    return new_mu, new_var

n_total = float(n_past + n_new)

# Combine mean of old and new data, taking into consideration
# (weighted) number of observations
total_mu = (n_new * new_mu + n_past * mu) / n_total

# Combine variance of old and new data, taking into consideration
# (weighted) number of observations. This is achieved by combining
# the sum-of-squared-differences (ssd)
old_ssd = n_past * var
new_ssd = n_new * new_var
total_ssd = (old_ssd + new_ssd +
             (n_new * n_past / n_total) * (mu - new_mu) ** 2)
total_var = total_ssd / n_total

return total_mu, total_var

def partial_fit(self, X, y, classes=None, sample_weight=None):
    """Incremental fit on a batch of samples.

    This method is expected to be called several times consecutively
    on different chunks of a dataset so as to implement out-of-core
    or online learning.

    This is especially useful when the whole dataset is too big to fit in
    memory at once.

    This method has some performance and numerical stability overhead,
    hence it is better to call partial_fit on chunks of data that are
    as large as possible (as long as fitting in the memory budget) to
    hide the overhead.

    Parameters
    -----
    X : array-like of shape (n_samples, n_features)
        Training vectors, where n_samples is the number of samples and
        n_features is the number of features.

```

```

y : array-like of shape (n_samples,)
    Target values.

classes : array-like of shape (n_classes,), default=None
    List of all the classes that can possibly appear in the y vector.

    Must be provided at the first call to partial_fit, can be omitted
    in subsequent calls.

sample_weight : array-like of shape (n_samples,), default=None
    Weights applied to individual samples (1. for unweighted).

    .. versionadded:: 0.17

Returns
-----
self : object
"""
return self._partial_fit(X, y, classes, _refit=False,
                        sample_weight=sample_weight)

def _partial_fit(self, X, y, classes=None, _refit=False,
                 sample_weight=None):
    """Actual implementation of Gaussian NB fitting.

Parameters
-----
X : array-like of shape (n_samples, n_features)
    Training vectors, where n_samples is the number of samples and
    n_features is the number of features.

y : array-like of shape (n_samples,)
    Target values.

classes : array-like of shape (n_classes,), default=None
    List of all the classes that can possibly appear in the y vector.

    Must be provided at the first call to partial_fit, can be omitted
    in subsequent calls.

    _refit : bool, default=False
        If true, act as though this were the first time we called
        _partial_fit (ie, throw away any past fitting and start over).

sample_weight : array-like of shape (n_samples,), default=None
    Weights applied to individual samples (1. for unweighted).

Returns
-----
self : object
"""
X, y = check_X_y(X, y)
if sample_weight is not None:
    sample_weight = _check_sample_weight(sample_weight, X)

# If the ratio of data variance between dimensions is too small, it
# will cause numerical errors. To address this, we artificially
# boost the variance by epsilon, a small fraction of the standard
# deviation of the largest dimension.
self.epsilon_ = self.var_smoothing * np.var(X, axis=0).max()

if __refit:

```

```

        self.classes_ = None

    if _check_partial_fit_first_call(self, classes):
        # This is the first call to partial_fit:
        # initialize various cumulative counters
        n_features = X.shape[1]
        n_classes = len(self.classes_)
        self.theta_ = np.zeros((n_classes, n_features))
        self.sigma_ = np.zeros((n_classes, n_features))

        self.class_count_ = np.zeros(n_classes, dtype=np.float64)

        # Initialise the class prior
        # Take into account the priors
        if self.priors is not None:
            priors = np.asarray(self.priors)
            # Check that the provide prior match the number of classes
            if len(priors) != n_classes:
                raise ValueError('Number of priors must match number of'
                                 ' classes.')
            # Check that the sum is 1
            if not np.isclose(priors.sum(), 1.0):
                raise ValueError('The sum of the priors should be 1.')
            # Check that the prior are non-negative
            if (priors < 0).any():
                raise ValueError('Priors must be non-negative.')
            self.class_prior_ = priors
        else:
            # Initialize the priors to zeros for each class
            self.class_prior_ = np.zeros(len(self.classes_),
                                         dtype=np.float64)
    else:
        if X.shape[1] != self.theta_.shape[1]:
            msg = "Number of features %d does not match previous data %d."
            raise ValueError(msg % (X.shape[1], self.theta_.shape[1]))
        # Put epsilon back in each time
        self.sigma_[:, :] -= self.epsilon_

    classes = self.classes_

    unique_y = np.unique(y)
    unique_y_in_classes = np.in1d(unique_y, classes)

    if not np.all(unique_y_in_classes):
        raise ValueError("The target label(s) %s in y do not exist in the "
                        "initial classes %s" %
                        (unique_y[~unique_y_in_classes], classes))

    for y_i in unique_y:
        i = classes.searchsorted(y_i)
        X_i = X[y == y_i, :]

        if sample_weight is not None:
            sw_i = sample_weight[y == y_i]
            N_i = sw_i.sum()
        else:
            sw_i = None
            N_i = X_i.shape[0]

        new_theta, new_sigma = self._update_mean_variance(
            self.class_count_[i], self.theta_[i, :], self.sigma_[i, :],
            X_i, sw_i)

```

```

        self.theta_[i, :] = new_theta
        self.sigma_[i, :] = new_sigma
        self.class_count_[i] += N_i

    self.sigma_[:, :] += self.epsilon_

    # Update if only no priors is provided
    if self.priors is None:
        # Empirical prior, with sample_weight taken into account
        self.class_prior_ = self.class_count_ / self.class_count_.sum()

    return self

def _joint_log_likelihood(self, X):
    joint_log_likelihood = []
    for i in range(np.size(self.classes_)):
        jointi = np.log(self.class_prior_[i])
        n_ij = - 0.5 * np.sum(np.log(2. * np.pi * self.sigma_[i, :]))
        n_ij -= 0.5 * np.sum(((X - self.theta_[i, :]) ** 2) /
                             (self.sigma_[i, :], 1))
        joint_log_likelihood.append(jointi + n_ij)

    joint_log_likelihood = np.array(joint_log_likelihood).T
    return joint_log_likelihood

_ALPHAMIN = 1e-10

class _BaseDiscreteNB(_BaseNB):
    """Abstract base class for naive Bayes on discrete/categorical data

    Any estimator based on this class should provide:

    _init_
    _joint_log_likelihood(X) as per _BaseNB
    """

    def _check_X(self, X):
        return check_array(X, accept_sparse='csr')

    def _check_X_y(self, X, y):
        return self._validate_data(X, y, accept_sparse='csr')

    def _update_class_log_prior(self, class_prior=None):
        n_classes = len(self.classes_)
        if class_prior is not None:
            if len(class_prior) != n_classes:
                raise ValueError("Number of priors must match number of"
                                 " classes.")
            self.class_log_prior_ = np.log(class_prior)
        elif self.fit_prior:
            with warnings.catch_warnings():
                # silence the warning when count is 0 because class was not yet
                # observed
                warnings.simplefilter("ignore", RuntimeWarning)
                log_class_count = np.log(self.class_count_)

                # empirical prior, with sample_weight taken into account
                self.class_log_prior_ = (log_class_count -
                                        np.log(self.class_count_.sum())))
        else:
            self.class_log_prior_ = np.full(n_classes, -np.log(n_classes))

```

```

def _check_alpha(self):
    if np.min(self.alpha) < 0:
        raise ValueError('Smoothing parameter alpha = %.1e. '
                         'alpha should be > 0.' % np.min(self.alpha))
    if isinstance(self.alpha, np.ndarray):
        if not self.alpha.shape[0] == self.n_features_:
            raise ValueError("alpha should be a scalar or a numpy array "
                             "with shape [n_features]")
    if np.min(self.alpha) < _ALPHA_MIN:
        warnings.warn('alpha too small will result in numeric errors, '
                      'setting alpha = %.1e' % _ALPHA_MIN)
        return np.maximum(self.alpha, _ALPHA_MIN)
    return self.alpha

def partial_fit(self, X, y, classes=None, sample_weight=None):
    """Incremental fit on a batch of samples.

    This method is expected to be called several times consecutively
    on different chunks of a dataset so as to implement out-of-core
    or online learning.

    This is especially useful when the whole dataset is too big to fit in
    memory at once.

    This method has some performance overhead hence it is better to call
    partial_fit on chunks of data that are as large as possible
    (as long as fitting in the memory budget) to hide the overhead.

    Parameters
    -----
    X : {array-like, sparse matrix} of shape (n_samples, n_features)
        Training vectors, where n_samples is the number of samples and
        n_features is the number of features.

    y : array-like of shape (n_samples,)
        Target values.

    classes : array-like of shape (n_classes), default=None
        List of all the classes that can possibly appear in the y vector.

        Must be provided at the first call to partial_fit, can be omitted
        in subsequent calls.

    sample_weight : array-like of shape (n_samples,), default=None
        Weights applied to individual samples (1. for unweighted).

    Returns
    -----
    self : object
    """
    X, y = self._check_X_y(X, y)
    _, n_features = X.shape

    if _check_partial_fit_first_call(self, classes):
        # This is the first call to partial_fit:
        # initialize various cumulative counters
        n_effective_classes = len(classes) if len(classes) > 1 else 2
        self._init_counters(n_effective_classes, n_features)
        self.n_features_ = n_features
    elif n_features != self.n_features_:
        msg = "Number of features %d does not match previous data %d."
        raise ValueError(msg % (n_features, self.n_features_))

```

```

Y = label_binarize(y, classes=self.classes_)
if Y.shape[1] == 1:
    Y = np.concatenate((1 - Y, Y), axis=1)

if X.shape[0] != Y.shape[0]:
    msg = "X.shape[0]=%d and y.shape[0]=%d are incompatible."
    raise ValueError(msg % (X.shape[0], y.shape[0]))

# label_binarize() returns arrays with dtype=np.int64.
# We convert it to np.float64 to support sample_weight consistently
Y = Y.astype(np.float64, copy=False)
if sample_weight is not None:
    sample_weight = _check_sample_weight(sample_weight, X)
    sample_weight = np.atleast_2d(sample_weight)
    Y *= sample_weight.T

class_prior = self.class_prior

# Count raw events from data before updating the class log prior
# and feature log probas
self._count(X, Y)

# XXX: OPTIM: we could introduce a public finalization method to
# be called by the user explicitly just once after several consecutive
# calls to partial_fit and prior any call to predict_[log_]proba]
# to avoid computing the smooth log probas at each call to partial fit
alpha = self._check_alpha()
self._update_feature_log_prob(alpha)
self._update_class_log_prior(class_prior=class_prior)
return self

def fit(self, X, y, sample_weight=None):
    """Fit Naive Bayes classifier according to X, y

    Parameters
    -----
    X : {array-like, sparse matrix} of shape (n_samples, n_features)
        Training vectors, where n_samples is the number of samples and
        n_features is the number of features.

    y : array-like of shape (n_samples,)
        Target values.

    sample_weight : array-like of shape (n_samples,), default=None
        Weights applied to individual samples (1. for unweighted).

    Returns
    -----
    self : object
    """
    X, y = self._check_X_y(X, y)
    _, n_features = X.shape
    self.n_features_ = n_features

    labelbin = LabelBinarizer()
    Y = labelbin.fit_transform(y)
    self.classes_ = labelbin.classes_
    if Y.shape[1] == 1:
        Y = np.concatenate((1 - Y, Y), axis=1)

    # LabelBinarizer().fit_transform() returns arrays with dtype=np.int64.
    # We convert it to np.float64 to support sample_weight consistently;

```

```

# this means we also don't have to cast X to floating point
if sample_weight is not None:
    Y = Y.astype(np.float64, copy=False)
    sample_weight = _check_sample_weight(sample_weight, X)
    sample_weight = np.atleast_2d(sample_weight)
    Y *= sample_weight.T

class_prior = self.class_prior

# Count raw events from data before updating the class log prior
# and feature log probas
n_effective_classes = Y.shape[1]

self._init_counters(n_effective_classes, n_features)
self._count(X, Y)
alpha = self._check_alpha()
self._update_feature_log_prob(alpha)
self._update_class_log_prior(class_prior=class_prior)
return self

def _init_counters(self, n_effective_classes, n_features):
    self.class_count_ = np.zeros(n_effective_classes, dtype=np.float64)
    self.feature_count_ = np.zeros((n_effective_classes, n_features),
                                   dtype=np.float64)

# XXX The following is a stopgap measure; we need to set the dimensions
# of class_log_prior_ and feature_log_prob_ correctly.
def _get_coef(self):
    return (self.feature_log_prob_[1:]
           if len(self.classes_) == 2 else self.feature_log_prob_)

def _get_intercept(self):
    return (self.class_log_prior_[1:]
           if len(self.classes_) == 2 else self.class_log_prior_)

coef_ = property(_get_coef)
intercept_ = property(_get_intercept)

def _more_tags(self):
    return {'poor_score': True}

class MultinomialNB(_BaseDiscreteNB):
    """
    Naive Bayes classifier for multinomial models

    The multinomial Naive Bayes classifier is suitable for classification with
    discrete features (e.g., word counts for text classification). The
    multinomial distribution normally requires integer feature counts. However,
    in practice, fractional counts such as tf-idf may also work.

    Read more in the :ref:`User Guide <multinomial_naive_bayes>`.
    """

    Parameters
    -----
    alpha : float, default=1.0
        Additive (Laplace/Lidstone) smoothing parameter
        (0 for no smoothing).

    fit_prior : bool, default=True
        Whether to learn class prior probabilities or not.
        If false, a uniform prior will be used.

```

```
class_prior : array-like of shape (n_classes,), default=None
    Prior probabilities of the classes. If specified the priors are not
    adjusted according to the data.

Attributes
-----
class_count_ : ndarray of shape (n_classes,)
    Number of samples encountered for each class during fitting. This
    value is weighted by the sample weight when provided.

class_log_prior_ : ndarray of shape (n_classes, )
    Smoothed empirical log probability for each class.

classes_ : ndarray of shape (n_classes,)
    Class labels known to the classifier

coef_ : ndarray of shape (n_classes, n_features)
    Mirrors ``feature_log_prob_`` for interpreting MultinomialNB
    as a linear model.

feature_count_ : ndarray of shape (n_classes, n_features)
    Number of samples encountered for each (class, feature)
    during fitting. This value is weighted by the sample weight when
    provided.

feature_log_prob_ : ndarray of shape (n_classes, n_features)
    Empirical log probability of features
    given a class, ``P(x_i|y)``.

intercept_ : ndarray of shape (n_classes, )
    Mirrors ``class_log_prior_`` for interpreting MultinomialNB
    as a linear model.

n_features_ : int
    Number of features of each sample.
```

Examples

```
>>> import numpy as np
>>> rng = np.random.RandomState(1)
>>> X = rng.randint(5, size=(6, 100))
>>> y = np.array([1, 2, 3, 4, 5, 6])
>>> from sklearn.naive_bayes import MultinomialNB
>>> clf = MultinomialNB()
>>> clf.fit(X, y)
MultinomialNB()
>>> print(clf.predict(X[2:3]))
[3]
```

Notes

For the rationale behind the names `coef_` and `intercept_`, i.e. naive Bayes as a linear classifier, see J. Rennie et al. (2003), Tackling the poor assumptions of naive Bayes text classifiers, ICML.

References

C.D. Manning, P. Raghavan and H. Schuetze (2008). Introduction to Information Retrieval. Cambridge University Press, pp. 234-265.
<https://nlp.stanford.edu/IR-book/html/htmledition/naive-bayes-text-classification-1.html>

```

@_deprecate_positional_args
def __init__(self, *, alpha=1.0, fit_prior=True, class_prior=None):
    self.alpha = alpha
    self.fit_prior = fit_prior
    self.class_prior = class_prior

def _more_tags(self):
    return {'requires_positive_X': True}

def _count(self, X, Y):
    """Count and smooth feature occurrences."""
    check_non_negative(X, "MultinomialNB (input X)")
    self.feature_count_ += safe_sparse_dot(Y.T, X)
    self.class_count_ += Y.sum(axis=0)

def _update_feature_log_prob(self, alpha):
    """Apply smoothing to raw counts and recompute log probabilities"""
    smoothed_fc = self.feature_count_ + alpha
    smoothed_cc = smoothed_fc.sum(axis=1)

    self.feature_log_prob_ = (np.log(smoothed_fc) -
                             np.log(smoothed_cc.reshape(-1, 1)))

def _joint_log_likelihood(self, X):
    """Calculate the posterior log probability of the samples X"""
    return (safe_sparse_dot(X, self.feature_log_prob_.T) +
            self.class_log_prior_)

class ComplementNB(_BaseDiscreteNB):
    """The Complement Naive Bayes classifier described in Rennie et al. (2003).

    The Complement Naive Bayes classifier was designed to correct the "severe
    assumptions" made by the standard Multinomial Naive Bayes classifier. It is
    particularly suited for imbalanced data sets.

    Read more in the :ref:`User Guide <complement_naive_bayes>`.

    Parameters
    -----
    alpha : float, default=1.0
        Additive (Laplace/Lidstone) smoothing parameter (0 for no smoothing).

    fit_prior : bool, default=True
        Only used in edge case with a single class in the training set.

    class_prior : array-like of shape (n_classes,), default=None
        Prior probabilities of the classes. Not used.

    norm : bool, default=False
        Whether or not a second normalization of the weights is performed. The
        default behavior mirrors the implementations found in Mahout and Weka,
        which do not follow the full algorithm described in Table 9 of the
        paper.

    Attributes
    -----
    class_count_ : ndarray of shape (n_classes,)
        Number of samples encountered for each class during fitting. This
        value is weighted by the sample weight when provided.

    class_log_prior_ : ndarray of shape (n_classes,)
        Smoothed empirical log probability for each class. Only used in edge

```

case with a single class in the training set.

```
classes_ : ndarray of shape (n_classes,)
    Class labels known to the classifier

feature_all_ : ndarray of shape (n_features,)
    Number of samples encountered for each feature during fitting. This
    value is weighted by the sample weight when provided.

feature_count_ : ndarray of shape (n_classes, n_features)
    Number of samples encountered for each (class, feature) during fitting.
    This value is weighted by the sample weight when provided.

feature_log_prob_ : ndarray of shape (n_classes, n_features)
    Empirical weights for class complements.

n_features_ : int
    Number of features of each sample.
```

Examples

```
-----
>>> import numpy as np
>>> rng = np.random.RandomState(1)
>>> X = rng.randint(5, size=(6, 100))
>>> y = np.array([1, 2, 3, 4, 5, 6])
>>> from sklearn.naive_bayes import ComplementNB
>>> clf = ComplementNB()
>>> clf.fit(X, y)
ComplementNB()
>>> print(clf.predict(X[2:3]))
[3]
```

References

```
-----
Rennie, J. D., Shih, L., Teevan, J., & Karger, D. R. (2003).
Tackling the poor assumptions of naive bayes text classifiers. In ICML
(Vol. 3, pp. 616-623).
https://people.csail.mit.edu/jrennie/papers/icml03-nb.pdf
"""
```

```
@_deprecate_positional_args
def __init__(self, *, alpha=1.0, fit_prior=True, class_prior=None,
            norm=False):
    self.alpha = alpha
    self.fit_prior = fit_prior
    self.class_prior = class_prior
    self.norm = norm

def __more_tags__(self):
    return {'requires_positive_X': True}

def __count__(self, X, Y):
    """Count feature occurrences."""
    check_non_negative(X, "ComplementNB (input X)")
    self.feature_count_ += safe_sparse_dot(Y.T, X)
    self.class_count_ += Y.sum(axis=0)
    self.feature_all_ = self.feature_count_.sum(axis=0)

def __update_feature_log_prob__(self, alpha):
    """Apply smoothing to raw counts and compute the weights."""
    comp_count = self.feature_all_ + alpha - self.feature_count_
    logged = np.log(comp_count / comp_count.sum(axis=1, keepdims=True))
    # BaseNB.predict uses argmax, but ComplementNB operates with argmin.
```

```

    if self.norm:
        summed = logged.sum(axis=1, keepdims=True)
        feature_log_prob = logged / summed
    else:
        feature_log_prob = -logged
    self.feature_log_prob_ = feature_log_prob

def _joint_log_likelihood(self, X):
    """Calculate the class scores for the samples in X."""
    jll = safe_sparse_dot(X, self.feature_log_prob_.T)
    if len(self.classes_) == 1:
        jll += self.class_log_prior_
    return jll

class BernoulliNB(_BaseDiscreteNB):
    """Naive Bayes classifier for multivariate Bernoulli models.

    Like MultinomialNB, this classifier is suitable for discrete data. The
    difference is that while MultinomialNB works with occurrence counts,
    BernoulliNB is designed for binary/boolean features.

    Read more in the :ref:`User Guide <beroulli_naive_bayes>`.

    Parameters
    -----
    alpha : float, default=1.0
        Additive (Laplace/Lidstone) smoothing parameter
        (0 for no smoothing).

    binarize : float or None, default=0.0
        Threshold for binarizing (mapping to booleans) of sample features.
        If None, input is presumed to already consist of binary vectors.

    fit_prior : bool, default=True
        Whether to learn class prior probabilities or not.
        If false, a uniform prior will be used.

    class_prior : array-like of shape (n_classes,), default=None
        Prior probabilities of the classes. If specified the priors are not
        adjusted according to the data.

    Attributes
    -----
    class_count_ : ndarray of shape (n_classes)
        Number of samples encountered for each class during fitting. This
        value is weighted by the sample weight when provided.

    class_log_prior_ : ndarray of shape (n_classes)
        Log probability of each class (smoothed).

    classes_ : ndarray of shape (n_classes,)
        Class labels known to the classifier

    feature_count_ : ndarray of shape (n_classes, n_features)
        Number of samples encountered for each (class, feature)
        during fitting. This value is weighted by the sample weight when
        provided.

    feature_log_prob_ : ndarray of shape (n_classes, n_features)
        Empirical log probability of features given a class,  $P(x_i | y)$ .

    n_features_ : int

```

Number of features of each sample.

Examples

```
-----
>>> import numpy as np
>>> rng = np.random.RandomState(1)
>>> X = rng.randint(5, size=(6, 100))
>>> Y = np.array([1, 2, 3, 4, 4, 5])
>>> from sklearn.naive_bayes import BernoulliNB
>>> clf = BernoulliNB()
>>> clf.fit(X, Y)
BernoulliNB()
>>> print(clf.predict(X[2:3]))
[3]
```

References

C.D. Manning, P. Raghavan and H. Schuetze (2008). Introduction to Information Retrieval. Cambridge University Press, pp. 234-265.
<https://nlp.stanford.edu/IR-book/html/htmledition/the-bernoulli-model-1.html>

A. McCallum and K. Nigam (1998). A comparison of event models for naive Bayes text classification. Proc. AAAI/ICML-98 Workshop on Learning for Text Categorization, pp. 41-48.

V. Metsis, I. Androutsopoulos and G. Palioras (2006). Spam filtering with naive Bayes -- Which naive Bayes? 3rd Conf. on Email and Anti-Spam (CEAS).
"""

```
@_deprecated_positional_args
def __init__(self, *, alpha=1.0, binarize=.0, fit_prior=True,
            class_prior=None):
    self.alpha = alpha
    self.binarize = binarize
    self.fit_prior = fit_prior
    self.class_prior = class_prior

def _check_X(self, X):
    X = super()._check_X(X)
    if self.binarize is not None:
        X = binarize(X, threshold=self.binarize)
    return X

def _check_X_y(self, X, y):
    X, y = super()._check_X_y(X, y)
    if self.binarize is not None:
        X = binarize(X, threshold=self.binarize)
    return X, y

def _count(self, X, Y):
    """Count and smooth feature occurrences."""
    self.feature_count_ += safe_sparse_dot(Y.T, X)
    self.class_count_ += Y.sum(axis=0)

def _update_feature_log_prob(self, alpha):
    """Apply smoothing to raw counts and recompute log probabilities"""
    smoothed_fc = self.feature_count_ + alpha
    smoothed_cc = self.class_count_ + alpha * 2

    self.feature_log_prob_ = (np.log(smoothed_fc) -
                             np.log(smoothed_cc.reshape(-1, 1)))

def _joint_log_likelihood(self, X):
```

```

"""Calculate the posterior log probability of the samples X"""
n_classes, n_features = self.feature_log_prob_.shape
n_samples, n_features_X = X.shape

if n_features_X != n_features:
    raise ValueError("Expected input with %d features, got %d instead"
                     % (n_features, n_features_X))

neg_prob = np.log(1 - np.exp(self.feature_log_prob_))
# Compute neg_prob · (1 - X).T as Σneg_prob - X · neg_prob
jll = safe_sparse_dot(X, (self.feature_log_prob_ - neg_prob).T)
jll += self.class_log_prior_ + neg_prob.sum(axis=1)

return jll


class CategoricalNB(_BaseDiscreteNB):
    """Naive Bayes classifier for categorical features

The categorical Naive Bayes classifier is suitable for classification with
discrete features that are categorically distributed. The categories of
each feature are drawn from a categorical distribution.

Read more in the :ref:`User Guide <categorical_naive_bayes>`.

Parameters
-----
alpha : float, default=1.0
    Additive (Laplace/Lidstone) smoothing parameter
    (0 for no smoothing).

fit_prior : bool, default=True
    Whether to learn class prior probabilities or not.
    If false, a uniform prior will be used.

class_prior : array-like of shape (n_classes,), default=None
    Prior probabilities of the classes. If specified the priors are not
    adjusted according to the data.

Attributes
-----
category_count_ : list of arrays of shape (n_features,)
    Holds arrays of shape (n_classes, n_categories of respective feature)
    for each feature. Each array provides the number of samples
    encountered for each class and category of the specific feature.

class_count_ : ndarray of shape (n_classes,)
    Number of samples encountered for each class during fitting. This
    value is weighted by the sample weight when provided.

class_log_prior_ : ndarray of shape (n_classes,)
    Smoothed empirical log probability for each class.

classes_ : ndarray of shape (n_classes,)
    Class labels known to the classifier

feature_log_prob_ : list of arrays of shape (n_features,)
    Holds arrays of shape (n_classes, n_categories of respective feature)
    for each feature. Each array provides the empirical log probability
    of categories given the respective feature and class, ``P(x_i|y)``.

n_features_ : int
    Number of features of each sample.

```

```

Examples
-----
>>> import numpy as np
>>> rng = np.random.RandomState(1)
>>> X = rng.randint(5, size=(6, 100))
>>> y = np.array([1, 2, 3, 4, 5, 6])
>>> from sklearn.naive_bayes import CategoricalNB
>>> clf = CategoricalNB()
>>> clf.fit(X, y)
CategoricalNB()
>>> print(clf.predict(X[2:3]))
[3]
"""

@_deprecate_positional_args
def __init__(self, *, alpha=1.0, fit_prior=True, class_prior=None):
    self.alpha = alpha
    self.fit_prior = fit_prior
    self.class_prior = class_prior

def fit(self, X, y, sample_weight=None):
    """Fit Naive Bayes classifier according to X, y

Parameters
-----
X : {array-like, sparse matrix} of shape (n_samples, n_features)
    Training vectors, where n_samples is the number of samples and
    n_features is the number of features. Here, each feature of X is
    assumed to be from a different categorical distribution.
    It is further assumed that all categories of each feature are
    represented by the numbers 0, ..., n - 1, where n refers to the
    total number of categories for the given feature. This can, for
    instance, be achieved with the help of OrdinalEncoder.

y : array-like of shape (n_samples,)
    Target values.

sample_weight : array-like of shape (n_samples), default=None
    Weights applied to individual samples (1. for unweighted).

Returns
-----
self : object
"""
    return super().fit(X, y, sample_weight=sample_weight)

def partial_fit(self, X, y, classes=None, sample_weight=None):
    """Incremental fit on a batch of samples.

This method is expected to be called several times consecutively
on different chunks of a dataset so as to implement out-of-core
or online learning.

This is especially useful when the whole dataset is too big to fit in
memory at once.

This method has some performance overhead hence it is better to call
partial_fit on chunks of data that are as large as possible
(as long as fitting in the memory budget) to hide the overhead.

Parameters
-----

```

```

X : {array-like, sparse matrix} of shape (n_samples, n_features)
    Training vectors, where n_samples is the number of samples and
    n_features is the number of features. Here, each feature of X is
    assumed to be from a different categorical distribution.
    It is further assumed that all categories of each feature are
    represented by the numbers 0, ..., n - 1, where n refers to the
    total number of categories for the given feature. This can, for
    instance, be achieved with the help of OrdinalEncoder.

y : array-like of shape (n_samples)
    Target values.

classes : array-like of shape (n_classes), default=None
    List of all the classes that can possibly appear in the y vector.

    Must be provided at the first call to partial_fit, can be omitted
    in subsequent calls.

sample_weight : array-like of shape (n_samples), default=None
    Weights applied to individual samples (1. for unweighted).

Returns
-----
self : object
"""
return super().partial_fit(X, y, classes,
                           sample_weight=sample_weight)

def _more_tags(self):
    return {'requires_positive_X': True}

def _check_X(self, X):
    X = check_array(X, dtype='int', accept_sparse=False,
                    force_all_finite=True)
    check_non_negative(X, "CategoricalNB (input X)")
    return X

def _check_X_y(self, X, y):
    X, y = self._validate_data(X, y, dtype='int', accept_sparse=False,
                               force_all_finite=True)
    check_non_negative(X, "CategoricalNB (input X)")
    return X, y

def _init_counters(self, n_effective_classes, n_features):
    self.class_count_ = np.zeros(n_effective_classes, dtype=np.float64)
    self.category_count_ = [np.zeros((n_effective_classes, 0))
                           for _ in range(n_features)]

def _count(self, X, Y):
    def _update_cat_count_dims(cat_count, highest_feature):
        diff = highest_feature + 1 - cat_count.shape[1]
        if diff > 0:
            # we append a column full of zeros for each new category
            return np.pad(cat_count, [(0, 0), (0, diff)], 'constant')
        return cat_count

    def _update_cat_count(X_feature, Y, cat_count, n_classes):
        for j in range(n_classes):
            mask = Y[:, j].astype(bool)
            if Y.dtype.type == np.int64:
                weights = None
            else:
                weights = Y[mask, j]
            cat_count[j] += weights
        return cat_count

    highest_feature = X.max()
    cat_count = self._count(X, Y)
    cat_count = _update_cat_count_dims(cat_count, highest_feature)
    cat_count = _update_cat_count(X_feature, Y, cat_count, n_classes)
    return cat_count

```

```

counts = np.bincount(X_feature[mask], weights=weights)
indices = np.nonzero(counts)[0]
cat_count[j, indices] += counts[indices]

self.class_count_ += Y.sum(axis=0)
for i in range(self.n_features_):
    X_feature = X[:, i]
    self.category_count_[i] = _update_cat_count_dims(
        self.category_count_[i], X_feature.max())
    _update_cat_count(X_feature, Y,
                      self.category_count_[i],
                      self.class_count_.shape[0])

def _update_feature_log_prob(self, alpha):
    feature_log_prob = []
    for i in range(self.n_features_):
        smoothed_cat_count = self.category_count_[i] + alpha
        smoothed_class_count = smoothed_cat_count.sum(axis=1)
        feature_log_prob.append(
            np.log(smoothed_cat_count) -
            np.log(smoothed_class_count.reshape(-1, 1)))
    self.feature_log_prob_ = feature_log_prob

def _joint_log_likelihood(self, X):
    if not X.shape[1] == self.n_features_:
        raise ValueError("Expected input with %d features, got %d instead"
                         % (self.n_features_, X.shape[1]))
    jll = np.zeros((X.shape[0], self.class_count_.shape[0]))
    for i in range(self.n_features_):
        indices = X[:, i]
        jll += self.feature_log_prob_[i][:, indices].T
    total_ll = jll + self.class_log_prior_
    return total_ll

# TODO: remove in 0.24
@deprecated("BaseNB is deprecated in version "
            "0.22 and will be removed in version 0.24.")
class BaseNB(_BaseNB):
    pass

# TODO: remove in 0.24
@deprecated("BaseDiscreteNB is deprecated in version "
            "0.22 and will be removed in version 0.24.")
class BaseDiscreteNB(_BaseDiscreteNB):
    pass

```

<https://github.com/pandas-dev/pandas/blob/master/pandas/core/frame.py#L370>

```
"""
DataFrame
-----
An efficient 2D container for potentially mixed-type time series or other
labeled data series.

Similar to its R counterpart, data.frame, except providing automatic data
alignment and a host of useful data manipulation methods having to do with the
labeling information
"""

import collections
from collections import abc
import datetime
from io import StringIO
import itertools
from textwrap import dedent
from typing import (
    IO,
    TYPE_CHECKING,
    Any,
    Dict,
    FrozenSet,
    Hashable,
    Iterable,
    Iterator,
    List,
    Optional,
    Sequence,
    Set,
    Tuple,
    Type,
    Union,
    cast,
)
import warnings

import numpy as np
import numpy.ma as ma

from pandas._config import get_option

from pandas._libs import algos as libalgos, lib, properties
from pandas._typing import (
    ArrayLike,
    Axes,
    Axis,
    Dtype,
    FilePathOrBuffer,
    Label,
    Level,
    Renamer,
)
from pandas.compat import PY37
from pandas.compat._optional import import_optional_dependency
from pandas.compat.numpy import function as nv
from pandas.util._decorators import (
```

```
Appender,
Substitution,
deprecate_kwarg,
doc,
rewrite_axis_style_signature,
)
from pandas.util._validators import (
    validate_axis_style_args,
    validate_bool_kwarg,
    validate_percentile,
)
from pandas.core.dtypes.cast import (
    cast_scalar_to_array,
    coerce_to_dtypes,
    find_common_type,
    infer_dtype_from_scalar,
    invalidate_string_dtypes,
    maybe_cast_to_datetime,
    maybe_convert_platform,
    maybe_downcast_to_dtype,
    maybe_infer_to_datetimelike,
    maybe_upcast,
    maybe_upcast_putmask,
    validate_numeric_casting,
)
from pandas.core.dtypes.common import (
    ensure_float64,
    ensure_int64,
    ensure_platform_int,
    infer_dtype_from_object,
    is_bool_dtype,
    is_dataclass,
    is_datetime64_any_dtype,
    is_dict_like,
    is_dtype_equal,
    is_extension_array_dtype,
    is_float_dtype,
    is_hashable,
    is_integer,
    is_integer_dtype,
    is_iterator,
    is_list_like,
    is_named_tuple,
    is_object_dtype,
    is_scalar,
    is_sequence,
    needs_i8_conversion,
    pandas_dtype,
)
from pandas.core.dtypes.generic import (
    ABCDataFrame,
    ABCIndexClass,
    ABCMultiIndex,
    ABCSeries,
)
from pandas.core.dtypes.missing import isna, notna

from pandas.core import algorithms, common as com, nanops, ops
from pandas.core.accessor import CachedAccessor
from pandas.core.arrays import Categorical, ExtensionArray
from pandas.core.arrays.datetimelike import DatetimeLikeArrayMixin as DatetimeLikeArray
from pandas.core.arrays.sparse import SparseFrameAccessor
```

```
from pandas.core.generic import NDFrame, _shared_docs
from pandas.core.indexes import base as ibase
from pandas.core.indexes.api import Index, ensure_index, ensure_index_from_sequences
from pandas.core.indexes.datetimes import DatetimeIndex
from pandas.core.indexes.multi import MultiIndex, maybe_droplevels
from pandas.core.indexes.period import PeriodIndex
from pandas.core.indexing import check_bool_indexer, convert_to_index_sliceable
from pandas.core.internals import BlockManager
from pandas.core.internals.construction import (
    arrays_to_mgr,
    dataclasses_to_dicts,
    get_names_from_index,
    init_dict,
    init_ndarray,
    masked_rec_array_to_mgr,
    reorder_arrays,
    sanitize_index,
    to_arrays,
)
from pandas.core.ops.missing import dispatch_fill_zeros
from pandas.core.series import Series

from pandas.io.common import get_filepath_or_buffer
from pandas.io.formats import console, format as fmt
from pandas.io.formats.info import info
import pandas.plotting

if TYPE_CHECKING:
    from pandas.core.groupby.generic import DataFrameGroupBy
    from pandas.io.formats.style import Styler

# -----
# Docstring templates

_shared_doc_kwargs = dict(
    axes="index, columns",
    klass="DataFrame",
    axes_single_arg="{0 or 'index', 1 or 'columns'}",
    axis="""axis : {0 or 'index', 1 or 'columns'}, default 0
        If 0 or 'index': apply function to each column.
        If 1 or 'columns': apply function to each row.""",
    optional_by="""
        by : str or list of str
            Name or list of names to sort by.

            - if `axis` is 0 or `index` then `by` may contain index
              levels and/or column labels.
            - if `axis` is 1 or `columns` then `by` may contain column
              levels and/or index labels.

        .. versionchanged:: 0.23.0

            Allow specifying index or column level names.""""",
    versionadded_to_excel="",
    optional_labels="""labels : array-like, optional
        New labels / index to conform the axis specified by 'axis' to.""""",
    optional_axis="""axis : int or str, optional
        Axis to target. Can be either the axis name ('index', 'columns')
        or number (0, 1).""""",
)
_numeric_only_doc = """numeric_only : boolean, default None
    Include only float, int, boolean data. If None, will attempt to use
```

```
    everything, then use only numeric data
"""

_merge_doc = """
Merge DataFrame or named Series objects with a database-style join.

The join is done on columns or indexes. If joining columns on
columns, the DataFrame indexes *will be ignored*. Otherwise if joining indexes
on indexes or indexes on a column or columns, the index will be passed on.

Parameters
-----
%s
right : DataFrame or named Series
    Object to merge with.
how : {'left', 'right', 'outer', 'inner'}, default 'inner'
    Type of merge to be performed.

    * left: use only keys from left frame, similar to a SQL left outer join;
      preserve key order.
    * right: use only keys from right frame, similar to a SQL right outer join;
      preserve key order.
    * outer: use union of keys from both frames, similar to a SQL full outer
      join; sort keys lexicographically.
    * inner: use intersection of keys from both frames, similar to a SQL inner
      join; preserve the order of the left keys.
on : label or list
    Column or index level names to join on. These must be found in both
    DataFrames. If `on` is None and not merging on indexes then this defaults
    to the intersection of the columns in both DataFrames.
left_on : label or list, or array-like
    Column or index level names to join on in the left DataFrame. Can also
    be an array or list of arrays of the length of the left DataFrame.
    These arrays are treated as if they are columns.
right_on : label or list, or array-like
    Column or index level names to join on in the right DataFrame. Can also
    be an array or list of arrays of the length of the right DataFrame.
    These arrays are treated as if they are columns.
left_index : bool, default False
    Use the index from the left DataFrame as the join key(s). If it is a
    MultiIndex, the number of keys in the other DataFrame (either the index
    or a number of columns) must match the number of levels.
right_index : bool, default False
    Use the index from the right DataFrame as the join key. Same caveats as
    left_index.
sort : bool, default False
    Sort the join keys lexicographically in the result DataFrame. If False,
    the order of the join keys depends on the join type (how keyword).
suffixes : tuple of (str, str), default ('_x', '_y')
    Suffix to apply to overlapping column names in the left and right
    side, respectively. To raise an exception on overlapping columns use
    (False, False).
copy : bool, default True
    If False, avoid copy if possible.
indicator : bool or str, default False
    If True, adds a column to output DataFrame called "_merge" with
    information on the source of each row.
    If string, column with information on source of each row will be added to
    output DataFrame, and column will be named value of string.
    Information column is Categorical-type and takes on a value of "left_only"
    for observations whose merge key only appears in 'left' DataFrame,
    "right_only" for observations whose merge key only appears in 'right'
    DataFrame, and "both" if the observation's merge key is found in both.
```

```
validate : str, optional
    If specified, checks if merge is of specified type.

    * "one_to_one" or "1:1": check if merge keys are unique in both
      left and right datasets.
    * "one_to_many" or "1:m": check if merge keys are unique in left
      dataset.
    * "many_to_one" or "m:1": check if merge keys are unique in right
      dataset.
    * "many_to_many" or "m:m": allowed, but does not result in checks.
```

Returns

DataFrame

A DataFrame of the two merged objects.

See Also

merge_ordered : Merge with optional filling/interpolation.

merge_asof : Merge on nearest keys.

DataFrame.join : Similar method using indices.

Notes

Support for specifying index levels as the `on`, `left_on`, and
`right_on` parameters was added in version 0.23.0

Support for merging named Series objects was added in version 0.24.0

Examples

```
>>> df1 = pd.DataFrame({'lkey': ['foo', 'bar', 'baz', 'foo'],
...                      'value': [1, 2, 3, 5]})
>>> df2 = pd.DataFrame({'rkey': ['foo', 'bar', 'baz', 'foo'],
...                      'value': [5, 6, 7, 8]})
>>> df1
   lkey  value
0  foo      1
1  bar      2
2  baz      3
3  foo      5
>>> df2
   rkey  value
0  foo      5
1  bar      6
2  baz      7
3  foo      8
```

Merge df1 and df2 on the lkey and rkey columns. The value columns have
the default suffixes, _x and _y, appended.

```
>>> df1.merge(df2, left_on='lkey', right_on='rkey')
   lkey  value_x  rkey  value_y
0  foo        1  foo        5
1  foo        1  foo        8
2  foo        5  foo        5
3  foo        5  foo        8
4  bar        2  bar        6
5  baz        3  baz        7
```

Merge DataFrames df1 and df2 with specified left and right suffixes
appended to any overlapping columns.

```
>>> df1.merge(df2, left_on='lkey', right_on='rkey',
```

```

...
    suffixes=('_left', '_right'))
lkey  value_left rkey  value_right
0   foo          1   foo          5
1   foo          1   foo          8
2   foo          5   foo          5
3   foo          5   foo          8
4   bar          2   bar          6
5   baz          3   baz          7

Merge DataFrames df1 and df2, but raise an exception if the DataFrames have
any overlapping columns.

>>> df1.merge(df2, left_on='lkey', right_on='rkey', suffixes=(False, False))
Traceback (most recent call last):
...
ValueError: columns overlap but no suffix specified:
    Index(['value'], dtype='object')
"""

# -----
# DataFrame class

class DataFrame(NDFrame):
    """
    Two-dimensional, size-mutable, potentially heterogeneous tabular data.

    Data structure also contains labeled axes (rows and columns).
    Arithmetic operations align on both row and column labels. Can be
    thought of as a dict-like container for Series objects. The primary
    pandas data structure.

    Parameters
    -----
    data : ndarray (structured or homogeneous), Iterable, dict, or DataFrame
        Dict can contain Series, arrays, constants, or list-like objects.

        .. versionchanged:: 0.23.0
            If data is a dict, column order follows insertion-order for
            Python 3.6 and later.

        .. versionchanged:: 0.25.0
            If data is a list of dicts, column order follows insertion-order
            for Python 3.6 and later.

    index : Index or array-like
        Index to use for resulting frame. Will default to RangeIndex if
        no indexing information part of input data and no index provided.
    columns : Index or array-like
        Column labels to use for resulting frame. Will default to
        RangeIndex (0, 1, 2, ..., n) if no column labels are provided.
    dtype : dtype, default None
        Data type to force. Only a single dtype is allowed. If None, infer.
    copy : bool, default False
        Copy data from inputs. Only affects DataFrame / 2d ndarray input.

    See Also
    -----
    DataFrame.from_records : Constructor from tuples, also record arrays.
    DataFrame.from_dict : From dicts of Series, arrays, or dicts.
    read_csv : Read a comma-separated values (csv) file into DataFrame.
    read_table : Read general delimited file into DataFrame.

```

```
read_clipboard : Read text from clipboard into DataFrame.
```

Examples

Constructing DataFrame from a dictionary.

```
>>> d = {'col1': [1, 2], 'col2': [3, 4]}
>>> df = pd.DataFrame(data=d)
>>> df
   col1  col2
0     1     3
1     2     4
```

Notice that the inferred dtype is int64.

```
>>> df.dtypes
col1    int64
col2    int64
dtype: object
```

To enforce a single dtype:

```
>>> df = pd.DataFrame(data=d, dtype=np.int8)
>>> df.dtypes
col1    int8
col2    int8
dtype: object
```

Constructing DataFrame from numpy ndarray:

```
>>> df2 = pd.DataFrame(np.array([[1, 2, 3], [4, 5, 6], [7, 8, 9]]),
...                      columns=['a', 'b', 'c'])
>>> df2
   a  b  c
0  1  2  3
1  4  5  6
2  7  8  9
"""
_internal_names_set = {"columns", "index"} | NDFrame._internal_names_set
_typ = "dataframe"

@property
def _constructor(self) -> Type["DataFrame"]:
    return DataFrame

_constructors_sliced: Type[Series] = Series
_deprecations: FrozenSet[str] = NDFrame._deprecations | frozenset({})
_accessors: Set[str] = {"sparse"}
```

@property

```
def _constructor_expanddim(self):
    # GH#31549 raising NotImplementedError on a property causes trouble
    # for `inspect`
    def constructor(*args, **kwargs):
        raise NotImplementedError("Not supported for DataFrames!")

    return constructor
```

```
# -----
# Constructors
```

```
def __init__(
```

```

self,
data=None,
index: Optional[Axes] = None,
columns: Optional[Axes] = None,
dtype: Optional[Dtype] = None,
copy: bool = False,
):
    if data is None:
        data = {}
    if dtype is not None:
        dtype = self._validate_dtype(dtype)

    if isinstance(data, DataFrame):
        data = data._mgr

    if isinstance(data, BlockManager):
        if index is None and columns is None and dtype is None and copy is False:
            # GH#33357 fastpath
            NDFrame.__init__(self, data)
            return

        mgr = self._init_mngr(
            data, axes=dict(index=index, columns=columns), dtype=dtype, copy=copy
        )
    elif isinstance(data, dict):
        mgr = init_dict(data, index, columns, dtype=dtype)
    elif isinstance(data, ma.MaskedArray):
        import numpy.ma.mrecords as mrecords

        # masked recarray
        if isinstance(data, mrecords.MaskedRecords):
            mgr = masked_rec_array_to_mngr(data, index, columns, dtype, copy)

        # a masked array
    else:
        mask = ma.getmaskarray(data)
        if mask.any():
            data, fill_value = maybe_upcast(data, copy=True)
            data.soften_mask() # set hardmask False if it was True
            data[mask] = fill_value
        else:
            data = data.copy()
        mgr = init_ndarray(data, index, columns, dtype=dtype, copy=copy)

    elif isinstance(data, (np.ndarray, Series, Index)):
        if data.dtype.names:
            data_columns = list(data.dtype.names)
            data = {k: data[k] for k in data_columns}
            if columns is None:
                columns = data_columns
            mgr = init_dict(data, index, columns, dtype=dtype)
        elif getattr(data, "name", None) is not None:
            mgr = init_dict({data.name: data}, index, columns, dtype=dtype)
        else:
            mgr = init_ndarray(data, index, columns, dtype=dtype, copy=copy)

    # For data is list-like, or Iterable (will consume into list)
    elif isinstance(data, abc.Iterable) and not isinstance(data, (str, bytes)):
        if not isinstance(data, (abc.Sequence, ExtensionArray)):
            data = list(data)
        if len(data) > 0:
            if is_dataclass(data[0]):
                data = dataclasses_to_dicts(data)

```

```

        if is_list_like(data[0]) and getattr(data[0], "ndim", 1) == 1:
            if is_named_tuple(data[0]) and columns is None:
                columns = data[0]._fields
            arrays, columns = to_arrays(data, columns, dtype=dtype)
            columns = ensure_index(columns)

            # set the index
            if index is None:
                if isinstance(data[0], Series):
                    index = get_names_from_index(data)
                elif isinstance(data[0], Categorical):
                    index = ibase.default_index(len(data[0]))
                else:
                    index = ibase.default_index(len(data))

            mgr = arrays_to_mgr(arrays, columns, index, columns, dtype=dtype)
        else:
            mgr = init_ndarray(data, index, columns, dtype=dtype, copy=copy)
    else:
        mgr = init_dict({}, index, columns, dtype=dtype)
else:
    try:
        arr = np.array(data, dtype=dtype, copy=copy)
    except (ValueError, TypeError) as err:
        exc = TypeError(
            "DataFrame constructor called with "
            f"incompatible data and dtype: {err}"
        )
        raise exc from err

    if arr.ndim == 0 and index is not None and columns is not None:
        values = cast_scalar_to_array(
            (len(index), len(columns)), data, dtype=dtype
        )
        mgr = init_ndarray(
            values, index, columns, dtype=values.dtype, copy=False
        )
    else:
        raise ValueError("DataFrame constructor not properly called!")

NDFrame.__init__(self, mgr)

```

```

@property
def axes(self) -> List[Index]:
    """
    Return a list representing the axes of the DataFrame.

```

It has the row axis labels and column axis labels as the only members. They are returned in that order.

Examples

```

>>> df = pd.DataFrame({'col1': [1, 2], 'col2': [3, 4]})
>>> df.axes
[RangeIndex(start=0, stop=2, step=1), Index(['col1', 'col2'],
dtype='object')]
"""
return [self.index, self.columns]

```

```

@property
def shape(self) -> Tuple[int, int]:

```

```
"""
Return a tuple representing the dimensionality of the DataFrame.

See Also
-----
ndarray.shape

Examples
-----
>>> df = pd.DataFrame({'col1': [1, 2], 'col2': [3, 4]})
>>> df.shape
(2, 2)

>>> df = pd.DataFrame({'col1': [1, 2], 'col2': [3, 4],
...                     'col3': [5, 6]})
>>> df.shape
(2, 3)
"""
return len(self.index), len(self.columns)

@property
def _is_homogeneous_type(self) -> bool:
    """
    Whether all the columns in a DataFrame have the same type.

    Returns
    -----
    bool

    See Also
    -----
    Index._is_homogeneous_type : Whether the object has a single
        dtype.
    MultiIndex._is_homogeneous_type : Whether all the levels of a
        MultiIndex have the same dtype.

    Examples
    -----
    >>> DataFrame({'A': [1, 2], 'B': [3, 4]})._is_homogeneous_type
    True
    >>> DataFrame({'A': [1, 2], 'B': [3.0, 4.0]})._is_homogeneous_type
    False

    Items with the same type but different sizes are considered
    different types.

    >>> DataFrame({
...     "A": np.array([1, 2], dtype=np.int32),
...     "B": np.array([1, 2], dtype=np.int64)})._is_homogeneous_type
    False
    """
    if self._mgr.any_extension_types:
        return len({block.dtype for block in self._mgr.blocks}) == 1
    else:
        return not self._mgr.is_mixed_type

@property
def _can_fast_transpose(self) -> bool:
    """
    Can we transpose this DataFrame without creating any new array objects.

    """
    if self._data.any_extension_types:
        # TODO(EA2D) special case would be unnecessary with 2D EAs
```

```

        return False
    return len(self._data.blocks) == 1

# -----
# Rendering Methods

def _repr_fits_vertical_(self) -> bool:
    """
    Check length against max_rows.
    """
    max_rows = get_option("display.max_rows")
    return len(self) <= max_rows

def _repr_fits_horizontal_(self, ignore_width: bool = False) -> bool:
    """
    Check if full repr fits in horizontal boundaries imposed by the display
    options width and max_columns.

    In case of non-interactive session, no boundaries apply.

    `ignore_width` is here so ipnb+HTML output can behave the way
    users expect. display.max_columns remains in effect.
    GH3541, GH3573
    """
    width, height = console.get_console_size()
    max_columns = get_option("display.max_columns")
    nb_columns = len(self.columns)

    # exceed max columns
    if (max_columns and nb_columns > max_columns) or (
        not ignore_width) and width and nb_columns > (width // 2)
    ):
        return False

    # used by repr_html under IPython notebook or scripts ignore terminal
    # dims
    if ignore_width or not console.in_interactive_session():
        return True

    if get_option("display.width") is not None or console.in_ipython_frontend():
        # check at least the column row for excessive width
        max_rows = 1
    else:
        max_rows = get_option("display.max_rows")

    # when auto-detecting, so width=None and not in ipython front end
    # check whether repr fits horizontal by actually checking
    # the width of the rendered repr
    buf = StringIO()

    # only care about the stuff we'll actually print out
    # and to_string on entire frame may be expensive
    d = self

    if not (max_rows is None): # unlimited rows
        # min of two, where one may be None
        d = d.iloc[: min(max_rows, len(d))]
    else:
        return True

    d.to_string(buf=buf)
    value = buf.getvalue()
    repr_width = max(len(l) for l in value.split("\n")))

```

```

        return repr_width < width

def _info_repr(self) -> bool:
    """
    True if the repr should show the info view.
    """
    info_repr_option = get_option("display.large_repr") == "info"
    return info_repr_option and not (
        self._repr_fits_horizontal_() and self._repr_fits_vertical_()
    )

def __repr__(self) -> str:
    """
    Return a string representation for a particular DataFrame.
    """
    buf = StringIO("")
    if self._info_repr():
        self.info(buf=buf)
        return buf.getvalue()

    max_rows = get_option("display.max_rows")
    min_rows = get_option("display.min_rows")
    max_cols = get_option("display.max_columns")
    max_colwidth = get_option("display.max_colwidth")
    show_dimensions = get_option("display.show_dimensions")
    if get_option("display.expand_frame_repr"):
        width, _ = console.get_console_size()
    else:
        width = None
    self.to_string(
        buf=buf,
        max_rows=max_rows,
        min_rows=min_rows,
        max_cols=max_cols,
        line_width=width,
        max_colwidth=max_colwidth,
        show_dimensions=show_dimensions,
    )
    return buf.getvalue()

def _repr_html_(self) -> Optional[str]:
    """
    Return a html representation for a particular DataFrame.

    Mainly for IPython notebook.
    """
    if self._info_repr():
        buf = StringIO("")
        self.info(buf=buf)
        # need to escape the <class>, should be the first line.
        val = buf.getvalue().replace("<", r"&lt;", 1)
        val = val.replace(">", r"&gt;", 1)
        return "<pre>" + val + "</pre>"

    if get_option("display.notebook_repr_html"):
        max_rows = get_option("display.max_rows")
        min_rows = get_option("display.min_rows")
        max_cols = get_option("display.max_columns")
        show_dimensions = get_option("display.show_dimensions")

        formatter = fmt.DataFrameFormatter(

```

```

        self,
        columns=None,
        col_space=None,
        na_rep="NaN",
        formatters=None,
        float_format=None,
        sparsify=None,
        justify=None,
        index_names=True,
        header=True,
        index=True,
        bold_rows=True,
        escape=True,
        max_rows=max_rows,
        min_rows=min_rows,
        max_cols=max_cols,
        show_dimensions=show_dimensions,
        decimal=".",
        table_id=None,
        render_links=False,
    )
    return formatter.to_html(notebook=True)
else:
    return None

@Substitution(
    header_type="bool or sequence",
    header="Write out the column names. If a list of strings "
    "is given, it is assumed to be aliases for the "
    "column names",
    col_space_type="int",
    col_space="The minimum width of each column",
)
@Substitution(shared_params=fmt.common_docstring, returns=fmt.return_docstring)
def to_string(
    self,
    buf: Optional[FilePathOrBuffer[str]] = None,
    columns: Optional[Sequence[str]] = None,
    col_space: Optional[int] = None,
    header: Union[bool, Sequence[str]] = True,
    index: bool = True,
    na_rep: str = "NaN",
    formatters: Optional[fmt.FormattersType] = None,
    float_format: Optional[fmt.FloatFormatType] = None,
    sparsify: Optional[bool] = None,
    index_names: bool = True,
    justify: Optional[str] = None,
    max_rows: Optional[int] = None,
    min_rows: Optional[int] = None,
    max_cols: Optional[int] = None,
    show_dimensions: bool = False,
    decimal: str = ".",
    line_width: Optional[int] = None,
    max_colwidth: Optional[int] = None,
    encoding: Optional[str] = None,
) -> Optional[str]:
    """
    Render a DataFrame to a console-friendly tabular output.
    %(shared_params)s
    line_width : int, optional
        Width to wrap a line in characters.
    max_colwidth : int, optional
        Max width to truncate each column in characters. By default, no limit.
    """

```

```

.. versionadded:: 1.0.0
encoding : str, default "utf-8"
    Set character encoding.

    .. versionadded:: 1.0
%(returns)s
See Also
-----
to_html : Convert DataFrame to HTML.

Examples
-----
>>> d = {'col1': [1, 2, 3], 'col2': [4, 5, 6]}
>>> df = pd.DataFrame(d)
>>> print(df.to_string())
   col1  col2
0      1      4
1      2      5
2      3      6
"""
from pandas import option_context

with option_context("display.max_colwidth", max_colwidth):
    formatter = fmt.DataFrameFormatter(
        self,
        columns=columns,
        col_space=col_space,
        na_rep=na_rep,
        formatters=formatters,
        float_format=float_format,
        sparsify=sparsify,
        justify=justify,
        index_names=index_names,
        header=header,
        index=index,
        min_rows=min_rows,
        max_rows=max_rows,
        max_cols=max_cols,
        show_dimensions=show_dimensions,
        decimal=decimal,
        line_width=line_width,
    )
    return formatter.to_string(buf=buf, encoding=encoding)

# -----
@property
def style(self) -> "Styler":
    """
    Returns a Styler object.

    Contains methods for building a styled HTML representation of the DataFrame.

    See Also
    -----
    io.formats.style.Styler : Helps style a DataFrame or Series according to the
        data with HTML and CSS.
    """
    from pandas.io.formats.style import Styler
    return Styler(self)

```

```

/shared_docs[
    "items"
] = r"""
Iterate over (column name, Series) pairs.

Iterates over the DataFrame columns, returning a tuple with
the column name and the content as a Series.

Yields
-----
label : object
    The column names for the DataFrame being iterated over.
content : Series
    The column entries belonging to each label, as a Series.

See Also
-----
DataFrame.iterrows : Iterate over DataFrame rows as
    (index, Series) pairs.
DataFrame.itertuples : Iterate over DataFrame rows as namedtuples
    of the values.

Examples
-----
>>> df = pd.DataFrame({'species': ['bear', 'bear', 'marsupial'],
...                      'population': [1864, 22000, 80000]},
...                     index=['panda', 'polar', 'koala'])
>>> df
      species  population
panda     bear        1864
polar     bear       22000
koala   marsupial     80000
>>> for label, content in df.items():
...     print(f'label: {label}')
...     print(f'content: {content}', sep='\n')
...
label: species
content:
panda         bear
polar         bear
koala   marsupial
Name: species, dtype: object
label: population
content:
panda        1864
polar       22000
koala       80000
Name: population, dtype: int64
"""

@appender(_shared_docs["items"])
def items(self) -> Iterable[Tuple[Label, Series]]:
    if self.columns.is_unique and hasattr(self, "_item_cache"):
        for k in self.columns:
            yield k, self._get_item_cache(k)
    else:
        for i, k in enumerate(self.columns):
            yield k, self._ixs(i, axis=1)

@appender(_shared_docs["items"])
def iteritems(self) -> Iterable[Tuple[Label, Series]]:
    yield from self.items()

```

```

def iterrows(self) -> Iterable[Tuple[Label, Series]]:
    """
    Iterate over DataFrame rows as (index, Series) pairs.

    Yields
    -----
    index : label or tuple of label
        The index of the row. A tuple for a `MultiIndex`.
    data : Series
        The data of the row as a Series.

    it : generator
        A generator that iterates over the rows of the frame.

    See Also
    -----
    DataFrame.itertuples : Iterate over DataFrame rows as namedtuples of the values.
    DataFrame.items : Iterate over (column name, Series) pairs.

    Notes
    -----
    1. Because ``iterrows`` returns a Series for each row,
       it does **not** preserve dtypes across the rows (dtypes are
       preserved across columns for DataFrames). For example,
       >>> df = pd.DataFrame([[1, 1.5]], columns=['int', 'float'])
       >>> row = next(df.iterrows())[1]
       >>> row
       int      1.0
       float    1.5
       Name: 0, dtype: float64
       >>> print(row['int'].dtype)
       float64
       >>> print(df['int'].dtype)
       int64
       To preserve dtypes while iterating over the rows, it is better
       to use :meth:`itertuples` which returns namedtuples of the values
       and which is generally faster than ``iterrows``.

    2. You should **never modify** something you are iterating over.
       This is not guaranteed to work in all cases. Depending on the
       data types, the iterator returns a copy and not a view, and writing
       to it will have no effect.
    """
    columns = self.columns
    klass = self._constructor_sliced
    for k, v in zip(self.index, self.values):
        s = klass(v, index=columns, name=k)
        yield k, s

def itertuples(self, index=True, name="Pandas"):
    """
    Iterate over DataFrame rows as namedtuples.

    Parameters
    -----
    index : bool, default True
        If True, return the index as the first element of the tuple.
    name : str or None, default "Pandas"
        The name of the returned namedtuples or None to return regular
        tuples.

```

```

>Returns
-----
iterator
    An object to iterate over namedtuples for each row in the
    DataFrame with the first field possibly being the index and
    following fields being the column values.

See Also
-----
DataFrame.iterrows : Iterate over DataFrame rows as (index, Series)
    pairs.
DataFrame.items : Iterate over (column name, Series) pairs.

Notes
-----
The column names will be renamed to positional names if they are
invalid Python identifiers, repeated, or start with an underscore.
On python versions < 3.7 regular tuples are returned for DataFrames
with a large number of columns (>254).

Examples
-----
>>> df = pd.DataFrame({'num_legs': [4, 2], 'num_wings': [0, 2]},
...                      index=['dog', 'hawk'])
>>> df
   num_legs  num_wings
dog          4          0
hawk         2          2
>>> for row in df.itertuples():
...     print(row)
...
Pandas(Index='dog', num_legs=4, num_wings=0)
Pandas(Index='hawk', num_legs=2, num_wings=2)

By setting the `index` parameter to False we can remove the index
as the first element of the tuple:

>>> for row in df.itertuples(index=False):
...     print(row)
...
Pandas(num_legs=4, num_wings=0)
Pandas(num_legs=2, num_wings=2)

With the `name` parameter set we set a custom name for the yielded
namedtuples:

>>> for row in df.itertuples(name='Animal'):
...     print(row)
...
Animal(Index='dog', num_legs=4, num_wings=0)
Animal(Index='hawk', num_legs=2, num_wings=2)
"""
arrays = []
fields = list(self.columns)
if index:
    arrays.append(self.index)
    fields.insert(0, "Index")

# use integer indexing because of possible duplicate column names
arrays.extend(self.iloc[:, k] for k in range(len(self.columns)))

# Python versions before 3.7 support at most 255 arguments to constructors
can_return_named_tuples = PY37 or len(self.columns) + index < 255

```

```

if name is not None and can_return_named_tuples:
    itertuple = collections.namedtuple(name, fields, rename=True)
    return map(itertuple._make, zip(*arrays))

# fallback to regular tuples
return zip(*arrays)

def __len__(self) -> int:
    """
    Returns length of info axis, but here we use the index.
    """
    return len(self.index)

def dot(self, other):
    """
    Compute the matrix multiplication between the DataFrame and other.

    This method computes the matrix product between the DataFrame and the
    values of an other Series, DataFrame or a numpy array.

    It can also be called using ``self @ other`` in Python >= 3.5.

    Parameters
    -----
    other : Series, DataFrame or array-like
        The other object to compute the matrix product with.

    Returns
    -----
    Series or DataFrame
        If other is a Series, return the matrix product between self and
        other as a Series. If other is a DataFrame or a numpy.array, return
        the matrix product of self and other in a DataFrame of a np.array.

    See Also
    -----
    Series.dot: Similar method for Series.

    Notes
    -----
    The dimensions of DataFrame and other must be compatible in order to
    compute the matrix multiplication. In addition, the column names of
    DataFrame and the index of other must contain the same values, as they
    will be aligned prior to the multiplication.

    The dot method for Series computes the inner product, instead of the
    matrix product here.

    Examples
    -----
    Here we multiply a DataFrame with a Series.

    >>> df = pd.DataFrame([[0, 1, -2, -1], [1, 1, 1, 1]])
    >>> s = pd.Series([1, 1, 2, 1])
    >>> df.dot(s)
    0      -4
    1       5
    dtype: int64

    Here we multiply a DataFrame with another DataFrame.

    >>> other = pd.DataFrame([[0, 1], [1, 2], [-1, -1], [2, 0]])
    >>> df.dot(other)

```

```
      0   1  
0   1   4  
1   2   2
```

Note that the dot method give the same result as @

```
>>> df @ other  
      0   1  
0   1   4  
1   2   2
```

The dot method works also if other is an np.array.

```
>>> arr = np.array([[0, 1], [1, 2], [-1, -1], [2, 0]])  
>>> df.dot(arr)  
      0   1  
0   1   4  
1   2   2
```

Note how shuffling of the objects does not change the result.

```
>>> s2 = s.reindex([1, 0, 2, 3])  
>>> df.dot(s2)  
0    -4  
1     5  
dtype: int64  
"""  
  
if isinstance(other, (Series, DataFrame)):  
    common = self.columns.union(other.index)  
    if len(common) > len(self.columns) or len(common) > len(other.index):  
        raise ValueError("matrices are not aligned")  
  
    left = self.reindex(columns=common, copy=False)  
    right = other.reindex(index=common, copy=False)  
    lvals = left.values  
    rvals = right.values  
else:  
    left = self  
    lvals = self.values  
    rvals = np.asarray(other)  
    if lvals.shape[1] != rvals.shape[0]:  
        raise ValueError(  
            f"Dot product shape mismatch, {lvals.shape} vs {rvals.shape}"  
        )  
  
if isinstance(other, DataFrame):  
    return self._constructor(  
        np.dot(lvals, rvals), index=left.index, columns=other.columns  
    )  
elif isinstance(other, Series):  
    return Series(np.dot(lvals, rvals), index=left.index)  
elif isinstance(rvals, (np.ndarray, Index)):  
    result = np.dot(lvals, rvals)  
    if result.ndim == 2:  
        return self._constructor(result, index=left.index)  
    else:  
        return Series(result, index=left.index)  
else: # pragma: no cover  
    raise TypeError(f"unsupported type: {type(other)}")  
  
def __matmul__(self, other):  
    """  
    Matrix multiplication using binary `@` operator in Python>=3.5.  
    """
```

```

"""
    return self.dot(other)

def __rmatmul__(self, other):
"""
Matrix multiplication using binary `@` operator in Python>=3.5.
"""
    return self.T.dot(np.transpose(other)).T

# -----
# IO methods (to / from other formats)

@classmethod
def from_dict(cls, data, orient="columns", dtype=None, columns=None) -> "DataFrame":
"""
Construct DataFrame from dict of array-like or dicts.

Creates DataFrame object from dictionary by columns or by index
allowing dtype specification.

Parameters
-----
data : dict
    Of the form {field : array-like} or {field : dict}.
orient : {'columns', 'index'}, default 'columns'
    The "orientation" of the data. If the keys of the passed dict
    should be the columns of the resulting DataFrame, pass 'columns'
    (default). Otherwise if the keys should be rows, pass 'index'.
dtype : dtype, default None
    Data type to force, otherwise infer.
columns : list, default None
    Column labels to use when ``orient='index'``. Raises a ValueError
    if used with ``orient='columns'``.

.. versionadded:: 0.23.0

Returns
-----
DataFrame

See Also
-----
DataFrame.from_records : DataFrame from ndarray (structured
    dtype), list of tuples, dict, or DataFrame.
DataFrame : DataFrame object creation using constructor.

Examples
-----
By default the keys of the dict become the DataFrame columns:

>>> data = {'col_1': [3, 2, 1, 0], 'col_2': ['a', 'b', 'c', 'd']}
>>> pd.DataFrame.from_dict(data)
   col_1  col_2
0      3      a
1      2      b
2      1      c
3      0      d

Specify ``orient='index'`` to create the DataFrame using dictionary
keys as rows:

>>> data = {'row_1': [3, 2, 1, 0], 'row_2': ['a', 'b', 'c', 'd']}
>>> pd.DataFrame.from_dict(data, orient='index')

```

```
      0  1  2  3
row_1  3  2  1  0
row_2  a  b  c  d
```

When using the 'index' orientation, the column names can be specified manually:

```
>>> pd.DataFrame.from_dict(data, orient='index',
...                           columns=['A', 'B', 'C', 'D'])
      A  B  C  D
row_1  3  2  1  0
row_2  a  b  c  d
"""
index = None
orient = orient.lower()
if orient == "index":
    if len(data) > 0:
        # TODO speed up Series case
        if isinstance(list(data.values())[0], (Series, dict)):
            data = _from_nested_dict(data)
        else:
            data, index = list(data.values()), list(data.keys())
elif orient == "columns":
    if columns is not None:
        raise ValueError("cannot use columns parameter with orient='columns'")
else: # pragma: no cover
    raise ValueError("only recognize index or columns for orient")

return cls(data, index=index, columns=columns, dtype=dtype)

def to_numpy(self, dtype=None, copy: bool = False) -> np.ndarray:
"""
Convert the DataFrame to a NumPy array.

.. versionadded:: 0.24.0
```

By default, the dtype of the returned array will be the common NumPy dtype of all types in the DataFrame. For example, if the dtypes are ``float16`` and ``float32``, the results dtype will be ``float32``. This may require copying data and coercing values, which may be expensive.

Parameters

`dtype` : str or numpy.dtype, optional

The dtype to pass to :meth:`numpy.asarray`.

`copy` : bool, default False

Whether to ensure that the returned value is a not a view on another array. Note that ``copy=False`` does not *ensure* that ``to_numpy()`` is no-copy. Rather, ``copy=True`` ensure that a copy is made, even if not strictly necessary.

Returns

numpy.ndarray

See Also

`Series.to_numpy` : Similar method for Series.

Examples

```
>>> pd.DataFrame({"A": [1, 2], "B": [3, 4]}).to_numpy()
```

```
array([[1, 3],  
       [2, 4]])
```

With heterogeneous data, the lowest common type will have to be used.

```
>>> df = pd.DataFrame({'A': [1, 2], "B": [3.0, 4.5]})  
>>> df.to_numpy()  
array([[1., 3.],  
       [2., 4.5]])
```

For a mix of numeric and non-numeric types, the output array will have object dtype.

```
>>> df['C'] = pd.date_range('2000', periods=2)  
>>> df.to_numpy()  
array([[1, 3.0, Timestamp('2000-01-01 00:00:00')],  
       [2, 4.5, Timestamp('2000-01-02 00:00:00')]], dtype=object)  
"""  
result = np.array(self.values, dtype=dtype, copy=copy)  
return result
```

```
def to_dict(self, orient="dict", into=dict):  
    """
```

Convert the DataFrame to a dictionary.

The type of the key-value pairs can be customized with the parameters (see below).

Parameters

orient : str {'dict', 'list', 'series', 'split', 'records', 'index'}
Determines the type of the values of the dictionary.

- 'dict' (default) : dict like {column -> {index -> value}}
- 'list' : dict like {column -> [values]}
- 'series' : dict like {column -> Series(values)}
- 'split' : dict like {'index' -> [index], 'columns' -> [columns], 'data' -> [values]}
- 'records' : list like [{column -> value}, ... , {column -> value}]
- 'index' : dict like {index -> {column -> value}}

Abbreviations are allowed. 's' indicates 'series' and 'sp' indicates 'split'.

into : class, default dict

The collections.abc.Mapping subclass used for all Mappings in the return value. Can be the actual class or an empty instance of the mapping type you want. If you want a collections.defaultdict, you must pass it initialized.

Returns

dict, list or collections.abc.Mapping

Return a collections.abc.Mapping object representing the DataFrame. The resulting transformation depends on the 'orient' parameter.

See Also

DataFrame.from_dict: Create a DataFrame from a dictionary.

DataFrame.to_json: Convert a DataFrame to JSON format.

Examples

```
>>> df = pd.DataFrame({'col1': [1, 2],
...                      'col2': [0.5, 0.75]},
...                      index=['row1', 'row2'])
>>> df
   col1  col2
row1    1  0.50
row2    2  0.75
>>> df.to_dict()
{'col1': {'row1': 1, 'row2': 2}, 'col2': {'row1': 0.5, 'row2': 0.75}}
```

You can specify the return orientation.

```
>>> df.to_dict('series')
{'col1': row1    1
 row2    2
Name: col1, dtype: int64,
'col2': row1    0.50
         row2    0.75
Name: col2, dtype: float64}

>>> df.to_dict('split')
{'index': ['row1', 'row2'], 'columns': ['col1', 'col2'],
 'data': [[1, 0.5], [2, 0.75]]}

>>> df.to_dict('records')
[{'col1': 1, 'col2': 0.5}, {'col1': 2, 'col2': 0.75}]

>>> df.to_dict('index')
{'row1': {'col1': 1, 'col2': 0.5}, 'row2': {'col1': 2, 'col2': 0.75}}
```

You can also specify the mapping type.

```
>>> from collections import OrderedDict, defaultdict
>>> df.to_dict(into=OrderedDict)
OrderedDict([('col1', OrderedDict([('row1', 1), ('row2', 2)])),
             ('col2', OrderedDict([('row1', 0.5), ('row2', 0.75)]))])
```

If you want a `defaultdict`, you need to initialize it:

```
>>> dd = defaultdict(list)
>>> df.to_dict('records', into=dd)
[defaultdict(<class 'list'>, {'col1': 1, 'col2': 0.5}),
 defaultdict(<class 'list'>, {'col1': 2, 'col2': 0.75})]
"""

if not self.columns.is_unique:
    warnings.warn(
        "DataFrame columns are not unique, some columns will be omitted.",
        UserWarning,
        stacklevel=2,
    )
# GH16122
into_c = com.standardize_mapping(into)

orient = orient.lower()
# GH32515
if orient.startswith(("d", "l", "s", "r", "i")) and orient not in {
    "dict",
    "list",
    "series",
    "split",
    "records",
```

```

    "index",
}:
    warnings.warn(
        "Using short name for 'orient' is deprecated. Only the "
        "'options: ('dict', list, 'series', 'split', 'records', 'index')' "
        "will be used in a future version. Use one of the above "
        "'to silence this warning.'",
        FutureWarning,
    )

    if orient.startswith("d"):
        orient = "dict"
    elif orient.startswith("l"):
        orient = "list"
    elif orient.startswith("sp"):
        orient = "split"
    elif orient.startswith("s"):
        orient = "series"
    elif orient.startswith("r"):
        orient = "records"
    elif orient.startswith("i"):
        orient = "index"

if orient == "dict":
    return into_c((k, v.to_dict(into)) for k, v in self.items())

elif orient == "list":
    return into_c((k, v.tolist()) for k, v in self.items())

elif orient == "split":
    return into_c(
        (
            ("index", self.index.tolist()),
            ("columns", self.columns.tolist()),
            (
                "data",
                [
                    list(map(com.maybe_box_datetimelike, t))
                    for t in self.itertuples(index=False, name=None)
                ],
            ),
        ),
    )

elif orient == "series":
    return into_c((k, com.maybe_box_datetimelike(v)) for k, v in self.items())

elif orient == "records":
    columns = self.columns.tolist()
    rows = (
        dict(zip(columns, row))
        for row in self.itertuples(index=False, name=None)
    )
    return [
        into_c((k, com.maybe_box_datetimelike(v)) for k, v in row.items())
        for row in rows
    ]

elif orient == "index":
    if not self.index.is_unique:
        raise ValueError("DataFrame index must be unique for orient='index'.")
    return into_c(
        (t[0], dict(zip(self.columns, t[1:]))))

```

```

        for t in self.itertuples(name=None)
    )

else:
    raise ValueError(f"orient '{orient}' not understood")

def to_gbq(
    self,
    destination_table,
    project_id=None,
    chunksize=None,
    reauth=False,
    if_exists="fail",
    auth_local_webserver=False,
    table_schema=None,
    location=None,
    progress_bar=True,
    credentials=None,
) -> None:
    """
    Write a DataFrame to a Google BigQuery table.

    This function requires the `pandas-gbq` package
    <https://pandas-gbq.readthedocs.io>__.

    See the `How to authenticate with Google BigQuery
    <https://pandas-gbq.readthedocs.io/en/latest/howto/authentication.html>__`_
    guide for authentication instructions.

    Parameters
    -----
    destination_table : str
        Name of table to be written, in the form ``dataset.tablename``.
    project_id : str, optional
        Google BigQuery Account project ID. Optional when available from
        the environment.
    chunksize : int, optional
        Number of rows to be inserted in each chunk from the dataframe.
        Set to ``None`` to load the whole dataframe at once.
    reauth : bool, default False
        Force Google BigQuery to re-authenticate the user. This is useful
        if multiple accounts are used.
    if_exists : str, default 'fail'
        Behavior when the destination table exists. Value can be one of:

        ``'fail'``
            If table exists raise pandas_gbq.TableCreationError.
        ``'replace'``
            If table exists, drop it, recreate it, and insert data.
        ``'append'``
            If table exists, insert data. Create if does not exist.
    auth_local_webserver : bool, default False
        Use the `local webserver flow`_ instead of the `console flow`_
        when getting user credentials.

        .. _local webserver flow:
            https://google-auth-
        oauthlib.readthedocs.io/en/latest/reference/google_auth_oauthlib.flow.html#google_auth_o
        authlib.flow.InstalledAppFlow.run_local_server
        .. _console flow:
            https://google-auth-
        oauthlib.readthedocs.io/en/latest/reference/google_auth_oauthlib.flow.html#google_auth_o
        authlib.flow.InstalledAppFlow.run_console
    
```

```
*New in version 0.2.0 of pandas-gbq*.
table_schema : list of dicts, optional
    List of BigQuery table fields to which according DataFrame
    columns conform to, e.g. ``[{'name': 'col1', 'type':
    'STRING'}, ...]``. If schema is not provided, it will be
    generated according to dtypes of DataFrame columns. See
    BigQuery API documentation on available names of a field.

*New in version 0.3.1 of pandas-gbq*.
location : str, optional
    Location where the load job should run. See the `BigQuery locations
    documentation
<https://cloud.google.com/bigquery/docs/dataset-locations>`__ for a
    list of available locations. The location must match that of the
    target dataset.

*New in version 0.5.0 of pandas-gbq*.
progress_bar : bool, default True
    Use the library `tqdm` to show the progress bar for the upload,
    chunk by chunk.

*New in version 0.5.0 of pandas-gbq*.
credentials : google.auth.credentials.Credentials, optional
    Credentials for accessing Google APIs. Use this parameter to
    override default credentials, such as to use Compute Engine
    :class:`google.auth.compute_engine.Credentials` or Service
    Account :class:`google.oauth2.service_account.Credentials`
    directly.

*New in version 0.8.0 of pandas-gbq*.

.. versionadded:: 0.24.0

See Also
-----
pandas_gbq.to_gbq : This function in the pandas-gbq library.
read_gbq : Read a DataFrame from Google BigQuery.
"""
from pandas.io import gbq

gbq.to_gbq(
    self,
    destination_table,
    project_id=project_id,
    chunksize=chunksize,
    reauth=reauth,
    if_exists=if_exists,
    auth_local_webserver=auth_local_webserver,
    table_schema=table_schema,
    location=location,
    progress_bar=progress_bar,
    credentials=credentials,
)

@classmethod
def from_records(
    cls,
    data,
    index=None,
    exclude=None,
    columns=None,
    coerce_float=False,
```

```
nrows=None,
) -> "DataFrame":
"""
Convert structured or record ndarray to DataFrame.

Parameters
-----
data : ndarray (structured dtype), list of tuples, dict, or DataFrame
index : str, list of fields, array-like
    Field of array to use as the index, alternately a specific set of
    input labels to use.
exclude : sequence, default None
    Columns or fields to exclude.
columns : sequence, default None
    Column names to use. If the passed data do not have names
    associated with them, this argument provides names for the
    columns. Otherwise this argument indicates the order of the columns
    in the result (any names not found in the data will become all-NA
    columns).
coerce_float : bool, default False
    Attempt to convert values of non-string, non-numeric objects (like
    decimal.Decimal) to floating point, useful for SQL result sets.
nrows : int, default None
    Number of rows to read if data is an iterator.

Returns
-----
DataFrame
"""
# Make a copy of the input columns so we can modify it
if columns is not None:
    columns = ensure_index(columns)

if is_iterator(data):
    if nrows == 0:
        return cls()

    try:
        first_row = next(data)
    except StopIteration:
        return cls(index=index, columns=columns)

    dtype = None
    if hasattr(first_row, "dtype") and first_row.dtype.names:
        dtype = first_row.dtype

    values = [first_row]

    if nrows is None:
        values += data
    else:
        values.extend(itertools.islice(data, nrows - 1))

    if dtype is not None:
        data = np.array(values, dtype=dtype)
    else:
        data = values

if isinstance(data, dict):
    if columns is None:
        columns = arr_columns = ensure_index(sorted(data))
        arrays = [data[k] for k in columns]
    else:
```

```

arrays = []
arr_columns = []
for k, v in data.items():
    if k in columns:
        arr_columns.append(k)
        arrays.append(v)

arrays, arr_columns = reorder_arrays(arrays, arr_columns, columns)

elif isinstance(data, (np.ndarray, DataFrame)):
    arrays, columns = to_arrays(data, columns)
    if columns is not None:
        columns = ensure_index(columns)
    arr_columns = columns
else:
    arrays, arr_columns = to_arrays(data, columns, coerce_float=coerce_float)

    arr_columns = ensure_index(arr_columns)
    if columns is not None:
        columns = ensure_index(columns)
    else:
        columns = arr_columns

if exclude is None:
    exclude = set()
else:
    exclude = set(exclude)

result_index = None
if index is not None:
    if isinstance(index, str) or not hasattr(index, "__iter__"):
        i = columns.get_loc(index)
        exclude.add(index)
        if len(arrays) > 0:
            result_index = Index(arrays[i], name=index)
        else:
            result_index = Index([], name=index)
    else:
        try:
            index_data = [arrays[arr_columns.get_loc(field)] for field in index]
        except (KeyError, TypeError):
            # raised by get_loc, see GH#29258
            result_index = index
        else:
            result_index = ensure_index_from_sequences(index_data, names=index)
            exclude.update(index)

if any(exclude):
    arr_exclude = [x for x in exclude if x in arr_columns]
    to_remove = [arr_columns.get_loc(col) for col in arr_exclude]
    arrays = [v for i, v in enumerate(arrays) if i not in to_remove]

    arr_columns = arr_columns.drop(arr_exclude)
    columns = columns.drop(exclude)

mgr = arrays_to_mgr(arrays, arr_columns, result_index, columns)

return cls(mgr)

def to_records(
    self, index=True, column_dtypes=None, index_dtypes=None
) -> np.recarray:
    """

```

```
Convert DataFrame to a NumPy record array.
```

```
Index will be included as the first field of the record array if requested.
```

Parameters

```
index : bool, default True  
    Include index in resulting record array, stored in 'index'  
    field or using the index label, if set.
```

```
column_dtypes : str, type, dict, default None  
    .. versionadded:: 0.24.0
```

```
If a string or type, the data type to store all columns. If a dictionary, a mapping of column names and indices (zero-indexed) to specific data types.
```

```
index_dtypes : str, type, dict, default None  
    .. versionadded:: 0.24.0
```

```
If a string or type, the data type to store all index levels. If a dictionary, a mapping of index level names and indices (zero-indexed) to specific data types.
```

```
This mapping is applied only if `index=True`.
```

Returns

```
numpy.recarray
```

```
NumPy ndarray with the DataFrame labels as fields and each row of the DataFrame as entries.
```

See Also

```
DataFrame.from_records: Convert structured or record ndarray to DataFrame.
```

```
numpy.recarray: An ndarray that allows field access using attributes, analogous to typed columns in a spreadsheet.
```

Examples

```
>>> df = pd.DataFrame({'A': [1, 2], 'B': [0.5, 0.75]},  
...                 index=['a', 'b'])  
>>> df  
      A      B  
a  1  0.50  
b  2  0.75  
>>> df.to_records()  
rec.array([('a', 1, 0.5), ('b', 2, 0.75)],  
         dtype=[('index', 'O'), ('A', '<i8'), ('B', '<f8')])
```

```
If the DataFrame index has no label then the recarray field name is set to 'index'. If the index has a label then this is used as the field name:
```

```
>>> df.index = df.index.rename("I")  
>>> df.to_records()  
rec.array([('a', 1, 0.5), ('b', 2, 0.75)],  
         dtype=[('I', 'O'), ('A', '<i8'), ('B', '<f8')])
```

```
The index can be excluded from the record array:
```

```
>>> df.to_records(index=False)
```

```

rec.array([(1, 0.5), (2, 0.75)],
          dtype=[('A', '<i8'), ('B', '<f8')])

Data types can be specified for the columns:

>>> df.to_records(column_dtypes={"A": "int32"})
rec.array([('a', 1, 0.5), ('b', 2, 0.75)],
          dtype=[('I', 'O'), ('A', '<i4'), ('B', '<f8')])

As well as for the index:

>>> df.to_records(index_dtypes=<S2")
rec.array([(b'a', 1, 0.5), (b'b', 2, 0.75)],
          dtype=[('I', 'S2'), ('A', '<i8'), ('B', '<f8')])

>>> index_dtypes = f"<S{df.index.str.len().max()}"
>>> df.to_records(index_dtypes=index_dtypes)
rec.array([(b'a', 1, 0.5), (b'b', 2, 0.75)],
          dtype=[('I', 'S1'), ('A', '<i8'), ('B', '<f8')])
"""

if index:
    if isinstance(self.index, ABCMultiIndex):
        # array of tuples to numpy cols. copy copy copy
        ix_vals = list(map(np.array, zip(*self.index.values)))
    else:
        ix_vals = [self.index.values]

    arrays = ix_vals + [
        np.asarray(self.iloc[:, i]) for i in range(len(self.columns))
    ]

    count = 0
    index_names = list(self.index.names)

    if isinstance(self.index, ABCMultiIndex):
        for i, n in enumerate(index_names):
            if n is None:
                index_names[i] = f"level_{count}"
            count += 1
    elif index_names[0] is None:
        index_names = ["index"]

    names = [str(name) for name in itertools.chain(index_names, self.columns)]
else:
    arrays = [np.asarray(self.iloc[:, i]) for i in range(len(self.columns))]
    names = [str(c) for c in self.columns]
    index_names = []

index_len = len(index_names)
formats = []

for i, v in enumerate(arrays):
    index = i

    # When the names and arrays are collected, we
    # first collect those in the DataFrame's index,
    # followed by those in its columns.
    #
    # Thus, the total length of the array is:
    # len(index_names) + len(DataFrame.columns).
    #
    # This check allows us to see whether we are
    # handling a name / array in the index or column.

```

```

        if index < index_len:
            dtype_mapping = index_dtypes
            name = index_names[index]
        else:
            index -= index_len
            dtype_mapping = column_dtypes
            name = self.columns[index]

        # We have a dictionary, so we get the data type
        # associated with the index or column (which can
        # be denoted by its name in the DataFrame or its
        # position in DataFrame's array of indices or
        # columns, whichever is applicable.
        if is_dict_like(dtype_mapping):
            if name in dtype_mapping:
                dtype_mapping = dtype_mapping[name]
            elif index in dtype_mapping:
                dtype_mapping = dtype_mapping[index]
            else:
                dtype_mapping = None

        # If no mapping can be found, use the array's
        # dtype attribute for formatting.
        #
        # A valid dtype must either be a type or
        # string naming a type.
        if dtype_mapping is None:
            formats.append(v.dtype)
        elif isinstance(dtype_mapping, (type, np.dtype, str)):
            formats.append(dtype_mapping)
        else:
            element = "row" if i < index_len else "column"
            msg = f"Invalid dtype {dtype_mapping} specified for {element} {name}"
            raise ValueError(msg)

    return np.rec.fromarrays(arrays, dtype={"names": names, "formats": formats})

@classmethod
def _from_arrays(
    cls,
    arrays,
    columns,
    index,
    dtype: Optional[Dtype] = None,
    verify_integrity: bool = True,
) -> "DataFrame":
    """
    Create DataFrame from a list of arrays corresponding to the columns.

    Parameters
    -----
    arrays : list-like of arrays
        Each array in the list corresponds to one column, in order.
    columns : list-like, Index
        The column names for the resulting DataFrame.
    index : list-like, Index
        The rows labels for the resulting DataFrame.
    dtype : dtype, optional
        Optional dtype to enforce for all arrays.
    verify_integrity : bool, default True
        Validate and homogenize all input. If set to False, it is assumed
        that all elements of `arrays` are actual arrays how they will be
        stored in a block (numpy ndarray or ExtensionArray), have the same
    """

```

```

length as and are aligned with the index, and that `columns` and
`index` are ensured to be an Index object.

>Returns
-----
DataFrame
"""
if dtype is not None:
    dtype = pandas_dtype(dtype)

mgr = arrays_to_mgr(
    arrays,
    columns,
    index,
    columns,
    dtype=dtype,
    verify_integrity=verify_integrity,
)
return cls(mgr)

@deprecate_kwarg(old_arg_name="fname", new_arg_name="path")
def to_stata(
    self,
    path: FilePathOrBuffer,
    convert_dates: Optional[Dict[Label, str]] = None,
    write_index: bool = True,
    byteorder: Optional[str] = None,
    time_stamp: Optional[datetime.datetime] = None,
    data_label: Optional[str] = None,
    variable_labels: Optional[Dict[Label, str]] = None,
    version: Optional[int] = 114,
    convert_strl: Optional[Sequence[Label]] = None,
) -> None:
    """
    Export DataFrame object to Stata dta format.

    Writes the DataFrame to a Stata dataset file.
    "dta" files contain a Stata dataset.

    Parameters
    -----
    path : str, buffer or path object
        String, path object (pathlib.Path or py._path.local.LocalPath) or
        object implementing a binary write() function. If using a buffer
        then the buffer will not be automatically closed after the file
        data has been written.

        .. versionchanged:: 1.0.0

    Previously this was "fname"

    convert_dates : dict
        Dictionary mapping columns containing datetime types to stata
        internal format to use when writing the dates. Options are 'tc',
        'td', 'tm', 'tw', 'th', 'tq', 'ty'. Column can be either an integer
        or a name. Datetime columns that do not have a conversion type
        specified will be converted to 'tc'. Raises NotImplementedError if
        a datetime column has timezone information.

    write_index : bool
        Write the index to Stata dataset.

    byteorder : str
        Can be ">", "<", "little", or "big". default is `sys.byteorder`.

    time_stamp : datetime

```

```
A datetime to use as file creation date. Default is the current
time.
data_label : str, optional
    A label for the data set. Must be 80 characters or smaller.
variable_labels : dict
    Dictionary containing columns as keys and variable labels as
    values. Each label must be 80 characters or smaller.
version : {114, 117, 118, 119, None}, default 114
    Version to use in the output dta file. Set to None to let pandas
    decide between 118 or 119 formats depending on the number of
    columns in the frame. Version 114 can be read by Stata 10 and
    later. Version 117 can be read by Stata 13 or later. Version 118
    is supported in Stata 14 and later. Version 119 is supported in
    Stata 15 and later. Version 114 limits string variables to 244
    characters or fewer while versions 117 and later allow strings
    with lengths up to 2,000,000 characters. Versions 118 and 119
    support Unicode characters, and version 119 supports more than
    32,767 variables.

.. versionadded:: 0.23.0
.. versionchanged:: 1.0.0

    Added support for formats 118 and 119.

convert_strl : list, optional
    List of column names to convert to string columns to Stata StrL
    format. Only available if version is 117. Storing strings in the
    StrL format can produce smaller dta files if strings have more than
    8 characters and values are repeated.

.. versionadded:: 0.23.0

Raises
-----
NotImplementedError
    * If datetimes contain timezone information
    * Column dtype is not representable in Stata
ValueError
    * Columns listed in convert_dates are neither datetime64[ns]
        or datetime.datetime
    * Column listed in convert_dates is not in DataFrame
    * Categorical label contains more than 32,000 characters

See Also
-----
read_stata : Import Stata data files.
io.stata.StataWriter : Low-level writer for Stata data files.
io.stata.StataWriter117 : Low-level writer for version 117 files.

Examples
-----
>>> df = pd.DataFrame({'animal': ['falcon', 'parrot', 'falcon',
...                                'parrot'],
...                      'speed': [350, 18, 361, 15]})
>>> df.to_stata('animals.dta') # doctest: +SKIP
"""
if version not in (114, 117, 118, 119, None):
    raise ValueError("Only formats 114, 117, 118 and 119 are supported.")
if version == 114:
    if convert_strl is not None:
        raise ValueError("strl is not supported in format 114")
    from pandas.io.stata import StataWriter as statawriter
elif version == 117:
```

```

        # mypy: Name 'statawriter' already defined (possibly by an import)
        from pandas.io.stata import StataWriter117 as statawriter # type: ignore
    else: # versions 118 and 119
        # mypy: Name 'statawriter' already defined (possibly by an import)
        from pandas.io.stata import StataWriterUTF8 as statawriter # type:ignore

    kwargs: Dict[str, Any] = {}
    if version is None or version >= 117:
        # strl conversion is only supported >= 117
        kwargs["convert_strl"] = convert_strl
    if version is None or version >= 118:
        # Specifying the version is only supported for UTF8 (118 or 119)
        kwargs["version"] = version

# mypy: Too many arguments for "StataWriter"
writer = statawriter( # type: ignore
    path,
    self,
    convert_dates=convert_dates,
    byteorder=byteorder,
    time_stamp=time_stamp,
    data_label=data_label,
    write_index=write_index,
    variable_labels=variable_labels,
    **kwargs,
)
writer.write_file()

@deprecate_kwarg(old_arg_name="fname", new_arg_name="path")
def to_feather(self, path, **kwargs) -> None:
    """
    Write a DataFrame to the binary Feather format.

    Parameters
    -----
    path : str
        String file path.
    **kwargs :
        Additional keywords passed to :func:`pyarrow.feather.write_feather`.
        Starting with pyarrow 0.17, this includes the `compression`,
        `compression_level`, `chunksize` and `version` keywords.

    .. versionadded:: 1.1.0
    """
    from pandas.io.feather_format import to_feather
    to_feather(self, path, **kwargs)

@Appender(
    """
    Examples
    -----
    >>> df = pd.DataFrame(
    ...     data={"animal_1": ["elk", "pig"], "animal_2": ["dog", "quetzal"]}
    ... )
    >>> print(df.to_markdown())
    |   | animal_1   | animal_2   |
    |---|:-----|:-----|
    | 0 | elk       | dog       |
    | 1 | pig       | quetzal  |
    """
)
@Substitution(klass="DataFrame")

```

```

@Appender(_shared_docs["to_markdown"])
def to_markdown(
    self, buf: Optional[IO[str]] = None, mode: Optional[str] = None, **kwargs
) -> Optional[str]:
    kwargs.setdefault("headers", "keys")
    kwargs.setdefault("tablefmt", "pipe")
    tabulate = import_optional_dependency("tabulate")
    result = tabulate.tabulate(self, **kwargs)
    if buf is None:
        return result
    buf, _, _ = get_filepath_or_buffer(buf, mode=mode)
    assert buf is not None # Help mypy.
    buf.writelines(result)
    return None

@deprecate_kwarg(old_arg_name="fname", new_arg_name="path")
def to_parquet(
    self,
    path,
    engine="auto",
    compression="snappy",
    index=None,
    partition_cols=None,
    **kwargs,
) -> None:
    """
    Write a DataFrame to the binary parquet format.

    This function writes the dataframe as a `parquet file
    <https://parquet.apache.org/>`_. You can choose different parquet
    backends, and have the option of compression. See
    :ref:`the user guide <io.parquet>` for more details.
    """

    Parameters
    -----
    path : str
        File path or Root Directory path. Will be used as Root Directory
        path while writing a partitioned dataset.

    .. versionchanged:: 1.0.0

        Previously this was "fname"

    engine : {'auto', 'pyarrow', 'fastparquet'}, default 'auto'
        Parquet library to use. If 'auto', then the option
        ``io.parquet.engine`` is used. The default ``io.parquet.engine``
        behavior is to try 'pyarrow', falling back to 'fastparquet' if
        'pyarrow' is unavailable.

    compression : {'snappy', 'gzip', 'brotli', None}, default 'snappy'
        Name of the compression to use. Use ``None`` for no compression.

    index : bool, default None
        If ``True``, include the dataframe's index(es) in the file output.
        If ``False``, they will not be written to the file.
        If ``None``, similar to ``True`` the dataframe's index(es)
        will be saved. However, instead of being saved as values,
        the RangeIndex will be stored as a range in the metadata so it
        doesn't require much space and is faster. Other indexes will
        be included as columns in the file output.

    .. versionadded:: 0.24.0

    partition_cols : list, optional, default None
        Column names by which to partition the dataset.

```

```
Columns are partitioned in the order they are given.
```

```
.. versionadded:: 0.24.0
```

**kwargs

```
Additional arguments passed to the parquet library. See  
:ref:`pandas io <io.parquet>` for more details.
```

See Also

```
-----
```

```
read_parquet : Read a parquet file.
```

```
DataFrame.to_csv : Write a csv file.
```

```
DataFrame.to_sql : Write to a sql table.
```

```
DataFrame.to_hdf : Write to hdf.
```

Notes

```
-----
```

```
This function requires either the `fastparquet`  
<https://pypi.org/project/fastparquet>`_ or `pyarrow`  
<https://arrow.apache.org/docs/python/>_ library.
```

Examples

```
-----
```

```
>>> df = pd.DataFrame(data={'col1': [1, 2], 'col2': [3, 4]})  
>>> df.to_parquet('df.parquet.gzip',  
...                 compression='gzip') # doctest: +SKIP  
>>> pd.read_parquet('df.parquet.gzip') # doctest: +SKIP  
    col1   col2  
0      1      3  
1      2      4  
"""
```

```
from pandas.io.parquet import to_parquet
```

to_parquet(

```
    self,  
    path,  
    engine,  
    compression=compression,  
    index=index,  
    partition_cols=partition_cols,  
    **kwargs,  
)
```

@Substitution(

```
    header_type="bool",  
    header="Whether to print column labels, default True",  
    col_space_type="str or int",  
    col_space="The minimum width of each column in CSS length "  
    "units. An int is assumed to be px units.\n\n"  
    "          .. versionadded:: 0.25.0\n"  
    "          Ability to use str",  
)
```

```
@Substitution(shared_params=fmt.common_docstring, returns=fmt.return_docstring)  
def to_html(  
    self,
```

```
    buf=None,  
    columns=None,  
    col_space=None,  
    header=True,  
    index=True,  
    na_rep="NaN",  
    formatters=None,  
    float_format=None,
```

```
sparsify=None,
index_names=True,
justify=None,
max_rows=None,
max_cols=None,
show_dimensions=False,
decimal=".",
bold_rows=True,
classes=None,
escape=True,
notebook=False,
border=None,
table_id=None,
render_links=False,
encoding=None,
):
"""
Render a DataFrame as an HTML table.
%(shared_params)s
bold_rows : bool, default True
    Make the row labels bold in the output.
classes : str or list or tuple, default None
    CSS class(es) to apply to the resulting html table.
escape : bool, default True
    Convert the characters <, >, and & to HTML-safe sequences.
notebook : {True, False}, default False
    Whether the generated HTML is for IPython Notebook.
border : int
    A ``border=border`` attribute is included in the opening
    `<table>` tag. Default ``pd.options.display.html.border``.
encoding : str, default "utf-8"
    Set character encoding.

.. versionadded:: 1.0

table_id : str, optional
    A css id is included in the opening `<table>` tag if specified.

.. versionadded:: 0.23.0

render_links : bool, default False
    Convert URLs to HTML links.

.. versionadded:: 0.24.0
%(returns)s
See Also
-----
to_string : Convert DataFrame to a string.
"""
if justify is not None and justify not in fmt._VALID_JUSTIFY_PARAMETERS:
    raise ValueError("Invalid value for justify parameter")

formatter = fmt.DataFrameFormatter(
    self,
    columns=columns,
    col_space=col_space,
    na_rep=na_rep,
    formatters=formatters,
    float_format=float_format,
    sparsify=sparsify,
    justify=justify,
    index_names=index_names,
    header=header,
```

```

    index=index,
    bold_rows=bold_rows,
    escape=escape,
    max_rows=max_rows,
    max_cols=max_cols,
    show_dimensions=show_dimensions,
    decimal=decimal,
    table_id=table_id,
    render_links=render_links,
)
# TODO: a generic formatter wld b in DataFrameFormatter
return formatter.to_html(
    buf=buf,
    classes=classes,
    notebook=notebook,
    border=border,
    encoding=encoding,
)

# -----
@Substitution(
    klass="DataFrame",
    type_sub=" and columns",
    max_cols_sub=(
        """max_cols : int, optional
        When to switch from the verbose to the truncated output. If the
        DataFrame has more than `max_cols` columns, the truncated output
        is used. By default, the setting in
        ``pandas.options.display.max_info_columns`` is used.
        """
),
    examples_sub=(
        """
>>> int_values = [1, 2, 3, 4, 5]
>>> text_values = ['alpha', 'beta', 'gamma', 'delta', 'epsilon']
>>> float_values = [0.0, 0.25, 0.5, 0.75, 1.0]
>>> df = pd.DataFrame({"int_col": int_values, "text_col": text_values,
...                      "float_col": float_values})
>>> df
   int_col  text_col  float_col
0         1    alpha      0.00
1         2     beta      0.25
2         3    gamma      0.50
3         4    delta      0.75
4         5  epsilon      1.00

Prints information of all columns:

>>> df.info(verbose=True)
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 5 entries, 0 to 4
Data columns (total 3 columns):
 #   Column      Non-Null Count  Dtype  
---  -- 
 0   int_col      5 non-null      int64  
 1   text_col     5 non-null      object 
 2   float_col    5 non-null      float64 
dtypes: float64(1), int64(1), object(1)
memory usage: 248.0+ bytes

Prints a summary of columns count and its dtypes but not per column
information:

```

```

>>> df.info(verbose=False)
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 5 entries, 0 to 4
Columns: 3 entries, int_col to float_col
dtypes: float64(1), int64(1), object(1)
memory usage: 248.0+ bytes

Pipe output of DataFrame.info to buffer instead of sys.stdout, get
buffer content and writes to a text file:

>>> import io
>>> buffer = io.StringIO()
>>> df.info(buf=buffer)
>>> s = buffer.getvalue()
>>> with open("df_info.txt", "w",
...             encoding="utf-8") as f: # doctest: +SKIP
...     f.write(s)
260

The `memory_usage` parameter allows deep introspection mode, specially
useful for big DataFrames and fine-tune memory optimization:

>>> random_strings_array = np.random.choice(['a', 'b', 'c'], 10 ** 6)
>>> df = pd.DataFrame({
...     'column_1': np.random.choice(['a', 'b', 'c'], 10 ** 6),
...     'column_2': np.random.choice(['a', 'b', 'c'], 10 ** 6),
...     'column_3': np.random.choice(['a', 'b', 'c'], 10 ** 6)
... })
>>> df.info()
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 1000000 entries, 0 to 999999
Data columns (total 3 columns):
 #   Column      Non-Null Count    Dtype  
---  -- 
 0   column_1    1000000 non-null   object 
 1   column_2    1000000 non-null   object 
 2   column_3    1000000 non-null   object 
dtypes: object(3)
memory usage: 22.9+ MB

>>> df.info(memory_usage='deep')
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 1000000 entries, 0 to 999999
Data columns (total 3 columns):
 #   Column      Non-Null Count    Dtype  
---  -- 
 0   column_1    1000000 non-null   object 
 1   column_2    1000000 non-null   object 
 2   column_3    1000000 non-null   object 
dtypes: object(3)
memory usage: 188.8 MB"""

),
see_also_sub=(
"""
    DataFrame.describe: Generate descriptive statistics of DataFrame
        columns.
    DataFrame.memory_usage: Memory usage of DataFrame columns."""
),
@doc(info)
def info(
    self,
    verbose: Optional[bool] = None,

```

```

buf: Optional[IO[str]] = None,
max_cols: Optional[int] = None,
memory_usage: Optional[Union[bool, str]] = None,
null_counts: Optional[bool] = None,
) -> None:
    return info(self, verbose, buf, max_cols, memory_usage, null_counts)

def memory_usage(self, index=True, deep=False) -> Series:
    """
    Return the memory usage of each column in bytes.

    The memory usage can optionally include the contribution of
    the index and elements of `object` dtype.

    This value is displayed in `DataFrame.info` by default. This can be
    suppressed by setting ``pandas.options.display.memory_usage`` to False.

    Parameters
    -----
    index : bool, default True
        Specifies whether to include the memory usage of the DataFrame's
        index in returned Series. If ``index=True``, the memory usage of
        the index is the first item in the output.
    deep : bool, default False
        If True, introspect the data deeply by interrogating
        `object` dtypes for system-level memory consumption, and include
        it in the returned values.

    Returns
    -----
    Series
        A Series whose index is the original column names and whose values
        is the memory usage of each column in bytes.

    See Also
    -----
    numpy.ndarray.nbytes : Total bytes consumed by the elements of an
        ndarray.
    Series.memory_usage : Bytes consumed by a Series.
    Categorical : Memory-efficient array for string values with
        many repeated values.
    DataFrame.info : Concise summary of a DataFrame.

    Examples
    -----
    >>> dtypes = ['int64', 'float64', 'complex128', 'object', 'bool']
    >>> data = dict([(t, np.ones(shape=5000).astype(t))
    ...           for t in dtypes])
    >>> df = pd.DataFrame(data)
    >>> df.head()
       int64   float64      complex128  object  bool
    0      1      1.0  1.000000+0.000000j      1  True
    1      1      1.0  1.000000+0.000000j      1  True
    2      1      1.0  1.000000+0.000000j      1  True
    3      1      1.0  1.000000+0.000000j      1  True
    4      1      1.0  1.000000+0.000000j      1  True

    >>> df.memory_usage()
    Index          128
    int64         40000
    float64       40000
    complex128    80000
    object         40000

```

```
bool          5000
dtype: int64

>>> df.memory_usage(index=False)
int64          40000
float64        40000
complex128     80000
object          40000
bool            5000
dtype: int64
```

The memory footprint of `object` dtype columns is ignored by default:

```
>>> df.memory_usage(deep=True)
Index           128
int64          40000
float64        40000
complex128     80000
object         160000
bool            5000
dtype: int64
```

Use a Categorical for efficient storage of an object-dtype column with many repeated values.

```
>>> df['object'].astype('category').memory_usage(deep=True)
5216
"""
result = Series(
    [c.memory_usage(index=False, deep=deep) for col, c in self.items()],
    index=self.columns,
)
if index:
    result = Series(self.index.memory_usage(deep=deep), index=["Index"]).append(
        result
    )
return result

def transpose(self, *args, copy: bool = False) -> "DataFrame":
    """
    Transpose index and columns.
```

Reflect the DataFrame over its main diagonal by writing rows as columns and vice-versa. The property :attr:`.T` is an accessor to the method :meth:`transpose`.

Parameters

*args : tuple, optional

Accepted for compatibility with NumPy.

copy : bool, default False

Whether to copy the data after transposing, even for DataFrames with a single dtype.

Note that a copy is always required for mixed dtype DataFrames, or for DataFrames with any extension types.

Returns

DataFrame

The transposed DataFrame.

See Also

```
-----  
numpy.transpose : Permute the dimensions of a given array.
```

Notes

```
-----  
Transposing a DataFrame with mixed dtypes will result in a homogeneous  
DataFrame with the `object` dtype. In such a case, a copy of the data  
is always made.
```

Examples

```
-----  
**Square DataFrame with homogeneous dtype**
```

```
>>> d1 = {'col1': [1, 2], 'col2': [3, 4]}  
>>> df1 = pd.DataFrame(data=d1)  
>>> df1  
      col1    col2  
0        1        3  
1        2        4  
  
>>> df1_transposed = df1.T # or df1.transpose()  
>>> df1_transposed  
      0  
col1  1  
col2  2  
      3  
      4
```

When the dtype is homogeneous in the original DataFrame, we get a transposed DataFrame with the same dtype:

```
>>> df1.dtypes  
col1    int64  
col2    int64  
dtype: object  
>>> df1_transposed.dtypes  
0    int64  
1    int64  
dtype: object
```

```
**Non-square DataFrame with mixed dtypes**
```

```
>>> d2 = {'name': ['Alice', 'Bob'],  
...         'score': [9.5, 8],  
...         'employed': [False, True],  
...         'kids': [0, 0]}  
>>> df2 = pd.DataFrame(data=d2)  
>>> df2  
      name   score  employed  kids  
0  Alice     9.5    False      0  
1    Bob      8.0    True      0  
  
>>> df2_transposed = df2.T # or df2.transpose()  
>>> df2_transposed  
      0      1  
name    Alice    Bob  
score     9.5      8  
employed  False   True  
kids        0      0
```

When the DataFrame has mixed dtypes, we get a transposed DataFrame with the `object` dtype:

```
>>> df2.dtypes  
name          object
```

```

score          float64
employed      bool
kids           int64
dtype: object
>>> df2_transposed.dtypes
0    object
1    object
dtype: object
"""
nv.validate_transpose(args, dict())
# construct the args

dtypes = list(self.dtypes)
if self._is_homogeneous_type and dtypes and is_extension_array_dtype(dtypes[0]):
    # We have EAs with the same dtype. We can preserve that dtype in transpose.
    dtype = dtypes[0]
    arr_type = dtype.construct_array_type()
    values = self.values

    new_values = [arr_type._from_sequence(row, dtype=dtype) for row in values]
    result = self._constructor(
        dict(zip(self.index, new_values)), index=self.columns
    )

else:
    new_values = self.values.T
    if copy:
        new_values = new_values.copy()
    result = self._constructor(
        new_values, index=self.columns, columns=self.index
    )

return result.__finalize__(self, method="transpose")

@property
def T(self) -> "DataFrame":
    return self.transpose()

# -----
# Indexing Methods

def __ixs(self, i: int, axis: int = 0):
    """
    Parameters
    -----
    i : int
    axis : int

    Notes
    -----
    If slice passed, the resulting data will be a view.
    """
    # irow
    if axis == 0:
        new_values = self._mgr.fast_xs(i)

        # if we are a copy, mark as such
        copy = isinstance(new_values, np.ndarray) and new_values.base is None
        result = self._constructor_sliced(
            new_values,
            index=self.columns,
            name=self.index[i],
            dtype=new_values.dtype,

```

```

        )
    result._set_is_copy(self, copy=copy)
    return result

    # i col
else:
    label = self.columns[i]

    values = self._mgr.iget(i)
    result = self._box_col_values(values, i)

    # this is a cached value, mark it so
    result._set_as_cached(label, self)

    return result

def _get_column_array(self, i: int) -> ArrayLike:
    """
    Get the values of the i'th column (ndarray or ExtensionArray, as stored
    in the Block)
    """
    return self._data.iget_values(i)

def _iter_column_arrays(self) -> Iterator[ArrayLike]:
    """
    Iterate over the arrays of all columns in order.
    This returns the values as stored in the Block (ndarray or ExtensionArray).
    """
    for i in range(len(self.columns)):
        yield self._get_column_array(i)

def __getitem__(self, key):
    key = lib.item_from_zerodim(key)
    key = com.apply_if_callable(key, self)

    if is_hashable(key):
        # shortcut if the key is in columns
        if self.columns.is_unique and key in self.columns:
            if self.columns.nlevels > 1:
                return self._getitem_multilevel(key)
            return self._get_item_cache(key)

    # Do we have a slicer (on rows)?
    indexer = convert_to_index_sliceable(self, key)
    if indexer is not None:
        # either we have a slice or we have a string that can be converted
        # to a slice for partial-string date indexing
        return self._slice(indexer, axis=0)

    # Do we have a (boolean) DataFrame?
    if isinstance(key, DataFrame):
        return self.where(key)

    # Do we have a (boolean) 1d indexer?
    if com.is_bool_indexer(key):
        return self._getitem_bool_array(key)

    # We are left with two options: a single key, and a collection of keys,
    # We interpret tuples as collections only for non-MultiIndex
    is_single_key = isinstance(key, tuple) or not is_list_like(key)

    if is_single_key:
        if self.columns.nlevels > 1:

```

```

        return self._getitem_multilevel(key)
    indexer = self.columns.get_loc(key)
    if is_integer(indexer):
        indexer = [indexer]
    else:
        if is_iterator(key):
            key = list(key)
        indexer = self.loc._get_listlike_indexer(key, axis=1, raise_missing=True)[1]

    # take() does not accept boolean indexers
    if getattr(indexer, "dtype", None) == bool:
        indexer = np.where(indexer)[0]

    data = self._take_with_is_copy(indexer, axis=1)

    if is_single_key:
        # What does looking for a single key in a non-unique index return?
        # The behavior is inconsistent. It returns a Series, except when
        # - the key itself is repeated (test on data.shape, #9519), or
        # - we have a MultiIndex on columns (test on self.columns, #21309)
        if data.shape[1] == 1 and not isinstance(self.columns, ABCMultiIndex):
            data = data[key]

    return data

def _getitem_bool_array(self, key):
    # also raises Exception if object array with NA values
    # warning here just in case -- previously __setitem__ was
    # reindexing but __getitem__ was not; it seems more reasonable to
    # go with the __setitem__ behavior since that is more consistent
    # with all other indexing behavior
    if isinstance(key, Series) and not key.index.equals(self.index):
        warnings.warn(
            "Boolean Series key will be reindexed to match DataFrame index.",
            UserWarning,
            stacklevel=3,
        )
    elif len(key) != len(self.index):
        raise ValueError(
            f"Item wrong length {len(key)} instead of {len(self.index)}."
        )

    # check_bool_indexer will throw exception if Series key cannot
    # be reindexed to match DataFrame rows
    key = check_bool_indexer(self.index, key)
    indexer = key.nonzero()[0]
    return self._take_with_is_copy(indexer, axis=0)

def _getitem_multilevel(self, key):
    # self.columns is a MultiIndex
    loc = self.columns.get_loc(key)
    if isinstance(loc, (slice, np.ndarray)):
        new_columns = self.columns[loc]
        result_columns = maybe_droplevels(new_columns, key)
        if self._is_mixed_type:
            result = self.reindex(columns=new_columns)
            result.columns = result_columns
        else:
            new_values = self.values[:, loc]
            result = self._constructor(
                new_values, index=self.index, columns=result_columns
            )
            result = result._finalize_(self)
    else:
        result = self._constructor(
            new_values, index=key, columns=new_columns
        )
    return result

```

```

# If there is only one column being returned, and its name is
# either an empty string, or a tuple with an empty string as its
# first element, then treat the empty string as a placeholder
# and return the column as if the user had provided that empty
# string in the key. If the result is a Series, exclude the
# implied empty string from its name.
if len(result.columns) == 1:
    top = result.columns[0]
    if isinstance(top, tuple):
        top = top[0]
    if top == "":
        result = result[""]
    if isinstance(result, Series):
        result = self._constructor_sliced(
            result, index=self.index, name=key
        )

    result._set_is_copy(self)
    return result
else:
    # loc is neither a slice nor ndarray, so must be an int
    return self._ixs(loc, axis=1)

def _get_value(self, index, col, takeable: bool = False):
    """
    Quickly retrieve single value at passed column and index.

    Parameters
    -----
    index : row label
    col : column label
    takeable : interpret the index/col as indexers, default False

    Returns
    -----
    scalar
    """
    if takeable:
        series = self._ixs(col, axis=1)
        return series._values[index]

    series = self._get_item_cache(col)
    engine = self.index._engine

    try:
        loc = engine.get_loc(index)
        return series._values[loc]
    except KeyError:
        # GH 20629
        if self.index.nlevels > 1:
            # partial indexing forbidden
            raise

    # we cannot handle direct indexing
    # use positional
    col = self.columns.get_loc(col)
    index = self.index.get_loc(index)
    return self._get_value(index, col, takeable=True)

def __setitem__(self, key, value):
    key = com.apply_if_callable(key, self)

```

```

# see if we can slice the rows
indexer = convert_to_index_sliceable(self, key)
if indexer is not None:
    # either we have a slice or we have a string that can be converted
    # to a slice for partial-string date indexing
    return self._setitem_slice(indexer, value)

if isinstance(key, DataFrame) or getattr(key, "ndim", None) == 2:
    self._setitem_frame(key, value)
elif isinstance(key, (Series, np.ndarray, list, Index)):
    self._setitem_array(key, value)
else:
    # set column
    self._set_item(key, value)

def _setitem_slice(self, key: slice, value):
    # NB: we can't just use self.loc[key] = value because that
    # operates on labels and we need to operate positional for
    # backwards-compat, xref GH#31469
    self._check_setitem_copy()
    self.iloc._setitem_with_indexer(key, value)

def _setitem_array(self, key, value):
    # also raises Exception if object array with NA values
    if com.is_bool_indexer(key):
        if len(key) != len(self.index):
            raise ValueError(
                f"Item wrong length {len(key)} instead of {len(self.index)}!")
        )
    key = check_bool_indexer(self.index, key)
    indexer = key.nonzero()[0]
    self._check_setitem_copy()
    self.iloc._setitem_with_indexer(indexer, value)
    else:
        if isinstance(value, DataFrame):
            if len(value.columns) != len(key):
                raise ValueError("Columns must be same length as key")
            for k1, k2 in zip(key, value.columns):
                self[k1] = value[k2]
        else:
            self.loc._ensure_listlike_indexer(key, axis=1)
            indexer = self.loc._get_listlike_indexer(
                key, axis=1, raise_missing=False
            )[1]
            self._check_setitem_copy()
            self.iloc._setitem_with_indexer((slice(None), indexer), value)

def _setitem_frame(self, key, value):
    # support boolean setting with DataFrame input, e.g.
    # df[df > df2] = 0
    if isinstance(key, np.ndarray):
        if key.shape != self.shape:
            raise ValueError("Array conditional must be same shape as self")
        key = self._constructor(key, **self._construct_axes_dict())

    if key.values.size and not is_bool_dtype(key.values):
        raise TypeError(
            "Must pass DataFrame or 2-d ndarray with boolean values only"
        )

    self._check_inplace_setting(value)
    self._check_setitem_copy()
    self._where(~key, value, inplace=True)

```

```

def _iset_item(self, loc: int, value):
    self._ensure_valid_index(value)

    # technically _sanitize_column expects a label, not a position,
    # but the behavior is the same as long as we pass broadcast=False
    value = self._sanitize_column(loc, value, broadcast=False)
    NDFrame._iset_item(self, loc, value)

    # check if we are modifying a copy
    # try to set first as we want an invalid
    # value exception to occur first
    if len(self):
        self._check_setitem_copy()

def _set_item(self, key, value):
    """
    Add series to DataFrame in specified column.

    If series is a numpy-array (not a Series/TimeSeries), it must be the
    same length as the DataFrames index or an error will be thrown.

    Series/TimeSeries will be conformed to the DataFrames index to
    ensure homogeneity.
    """
    self._ensure_valid_index(value)
    value = self._sanitize_column(key, value)
    NDFrame._set_item(self, key, value)

    # check if we are modifying a copy
    # try to set first as we want an invalid
    # value exception to occur first
    if len(self):
        self._check_setitem_copy()

def _set_value(self, index, col, value, takeable: bool = False):
    """
    Put single value at passed column and index.

    Parameters
    -----
    index : row label
    col : column label
    value : scalar
    takeable : interpret the index/col as indexers, default False
    """
    try:
        if takeable is True:
            series = self._ixs(col, axis=1)
            series._set_value(index, value, takeable=True)
            return

        series = self._get_item_cache(col)
        engine = self.index._engine
        loc = engine.get_loc(index)
        validate_numeric_casting(series.dtype, value)

        series._values[loc] = value
        # Note: trying to use series._set_value breaks tests in
        # tests.frame.indexing.test_indexing and tests.indexing.test_partial
    except (KeyError, TypeError):
        # set using a non-recursive method & reset the cache
        if takeable:

```

```

        self.iloc[index, col] = value
    else:
        self.loc[index, col] = value
    self._item_cache.pop(col, None)

def _ensure_valid_index(self, value):
    """
    Ensure that if we don't have an index, that we can create one from the
    passed value.
    """
    # GH5632, make sure that we are a Series convertible
    if not len(self.index) and is_list_like(value) and len(value):
        try:
            value = Series(value)
        except (ValueError, NotImplemented, TypeError) as err:
            raise ValueError(
                "Cannot set a frame with no defined index "
                "and a value that cannot be converted to a Series"
            ) from err

        self._mgr = self._mgr.reindex_axis(
            value.index.copy(), axis=1, fill_value=np.nan
        )

def _box_col_values(self, values, loc: int) -> Series:
    """
    Provide boxed values for a column.
    """
    # Lookup in columns so that if e.g. a str datetime was passed
    # we attach the Timestamp object as the name.
    name = self.columns[loc]
    klass = self._constructor_sliced
    return klass(values, index=self.index, name=name, fastpath=True)

# -----
# Unsorted

def query(self, expr, inplace=False, **kwargs):
    """
    Query the columns of a DataFrame with a boolean expression.

    Parameters
    -----
    expr : str
        The query string to evaluate.

        You can refer to variables
        in the environment by prefixing them with an '@' character like
        ``@a + b``.

        You can refer to column names that contain spaces or operators by
        surrounding them in backticks. This way you can also escape
        names that start with a digit, or those that are a Python keyword.
        Basically when it is not valid Python identifier. See notes down
        for more details.

        For example, if one of your columns is called ``a a`` and you want
        to sum it with ``b``, your query should be ```a a` + b``.

        .. versionadded:: 0.25.0
            Backtick quoting introduced.

        .. versionadded:: 1.0.0
    """

```

Expanding functionality of backtick quoting for more than only spaces.

```
inplace : bool  
    Whether the query should modify the data in place or return  
    a modified copy.  
**kwargs  
    See the documentation for :func:`eval` for complete details  
    on the keyword arguments accepted by :meth:`DataFrame.query`.
```

Returns

DataFrame
 DataFrame resulting from the provided query expression.

See Also

```
-----  
eval : Evaluate a string describing operations on  
    DataFrame columns.  
DataFrame.eval : Evaluate a string describing operations on  
    DataFrame columns.
```

Notes

```
-----  
The result of the evaluation of this expression is first passed to  
:attr:`DataFrame.loc` and if that fails because of a  
multidimensional key (e.g., a DataFrame) then the result will be passed  
to :meth:`DataFrame.__getitem__`.
```

This method uses the top-level :func:`eval` function to
evaluate the passed query.

The :meth:`~pandas.DataFrame.query` method uses a slightly
modified Python syntax by default. For example, the ``&`` and ``|``
(bitwise) operators have the precedence of their boolean cousins,
:keyword:`and` and :keyword:`or`. This **is** syntactically valid Python,
however the semantics are different.

You can change the semantics of the expression by passing the keyword
argument ``parser='python'``. This enforces the same semantics as
evaluation in Python space. Likewise, you can pass ``engine='python'``
to evaluate an expression using Python itself as a backend. This is not
recommended as it is inefficient compared to using ``numexpr`` as the
engine.

The :attr:`DataFrame.index` and
:attr:`DataFrame.columns` attributes of the
:class:`~pandas.DataFrame` instance are placed in the query namespace
by default, which allows you to treat both the index and columns of the
frame as a column in the frame.
The identifier ``index`` is used for the frame index; you can also
use the name of the index to identify it in a query. Please note that
Python keywords may not be used as identifiers.

For further details and examples see the ``query`` documentation in
:ref:`indexing <indexing.query>`.

Backtick quoted variables

Backtick quoted variables are parsed as literal Python code and
are converted internally to a Python valid identifier.
This can lead to the following problems.

During parsing a number of disallowed characters inside the backtick

quoted string are replaced by strings that are allowed as a Python identifier. These characters include all operators in Python, the space character, the question mark, the exclamation mark, the dollar sign, and the euro sign. For other characters that fall outside the ASCII range (U+0001..U+007F) and those that are not further specified in PEP 3131, the query parser will raise an error.

This excludes whitespace different than the space character, but also the hashtag (as it is used for comments) and the backtick itself (backtick can also not be escaped).

In a special case, quotes that make a pair around a backtick can confuse the parser.

For example, ``it's` > `that's`` will raise an error, as it forms a quoted string (''s > `that``) with a backtick inside.

See also the Python documentation about lexical analysis (https://docs.python.org/3/reference/lexical_analysis.html) in combination with the source code in :mod:`pandas.core.computation.parsing`.

Examples

```
>>> df = pd.DataFrame({'A': range(1, 6),
...                      'B': range(10, 0, -2),
...                      'C C': range(10, 5, -1)})
>>> df
   A    B    C C
0  1    10   10
1  2     8    9
2  3     6    8
3  4     4    7
4  5     2    6
>>> df.query('A > B')
   A    B    C C
4  5     2    6
```

The previous expression is equivalent to

```
>>> df[df.A > df.B]
   A    B    C C
4  5     2    6
```

For columns with spaces in their name, you can use backtick quoting.

```
>>> df.query('B == `C C`')
   A    B    C C
0  1    10   10
```

The previous expression is equivalent to

```
>>> df[df.B == df['C C']]
   A    B    C C
0  1    10   10
"""
inplace = validate_bool_kwarg(inplace, "inplace")
if not isinstance(expr, str):
    msg = f"expr must be a string to be evaluated, {type(expr)} given"
    raise ValueError(msg)
kwargs["level"] = kwargs.pop("level", 0) + 1
kwargs["target"] = None
res = self.eval(expr, **kwargs)

try:
    result = self.loc[res]
```

```
except ValueError:
    # when res is multi-dimensional loc raises, but this is sometimes a
    # valid query
    result = self[res]

if inplace:
    self._update_inplace(result)
else:
    return result

def eval(self, expr, inplace=False, **kwargs):
    """
    Evaluate a string describing operations on DataFrame columns.

    Operates on columns only, not specific rows or elements. This allows
    `eval` to run arbitrary code, which can make you vulnerable to code
    injection if you pass user input to this function.

    Parameters
    -----
    expr : str
        The expression string to evaluate.
    inplace : bool, default False
        If the expression contains an assignment, whether to perform the
        operation inplace and mutate the existing DataFrame. Otherwise,
        a new DataFrame is returned.
    **kwargs
        See the documentation for :func:`eval` for complete details
        on the keyword arguments accepted by
        :meth:`~pandas.DataFrame.query`.

    Returns
    -----
    ndarray, scalar, or pandas object
        The result of the evaluation.

    See Also
    -----
    DataFrame.query : Evaluates a boolean expression to query the columns
        of a frame.
    DataFrame.assign : Can evaluate an expression or function to create new
        values for a column.
    eval : Evaluate a Python expression as a string using various
        backends.

    Notes
    -----
    For more details see the API documentation for :func:`~eval`.
    For detailed examples see :ref:`enhancing performance with eval
    <enhancingperf.eval>`.

    Examples
    -----
    >>> df = pd.DataFrame({'A': range(1, 6), 'B': range(10, 0, -2)})
    >>> df
       A   B
    0  1  10
    1  2   8
    2  3   6
    3  4   4
    4  5   2
    >>> df.eval('A + B')
    0    11
```

```
1    10
2     9
3     8
4     7
dtype: int64
```

Assignment is allowed though by default the original DataFrame is not modified.

```
>>> df.eval('C = A + B')
   A   B   C
0  1  10  11
1  2    8  10
2  3    6    9
3  4    4    8
4  5    2    7
>>> df
   A   B
0  1  10
1  2    8
2  3    6
3  4    4
4  5    2
```

Use ``inplace=True`` to modify the original DataFrame.

```
>>> df.eval('C = A + B', inplace=True)
>>> df
   A   B   C
0  1  10  11
1  2    8  10
2  3    6    9
3  4    4    8
4  5    2    7
```

Multiple columns can be assigned to using multi-line expressions:

```
>>> df.eval(
...     '''
...     C = A + B
...     D = A - B
...     '''
... )
   A   B   C   D
0  1  10  11 -9
1  2    8  10 -6
2  3    6    9 -3
3  4    4    8  0
4  5    2    7  3
"""
from pandas.core.computation.eval import eval as _eval

inplace = validate_bool_kwarg(inplace, "inplace")
resolvers = kwargs.pop("resolvers", None)
kwargs["level"] = kwargs.pop("level", 0) + 1
if resolvers is None:
    index_resolvers = self._get_index_resolvers()
    column_resolvers = self._get_cleaned_column_resolvers()
    resolvers = column_resolvers, index_resolvers
if "target" not in kwargs:
    kwargs["target"] = self
kwargs["resolvers"] = kwargs.get("resolvers", ()) + tuple(resolvers)
```

```

    return _eval(expr, inplace=inplace, **kwargs)

def select_dtypes(self, include=None, exclude=None) -> "DataFrame":
    """
    Return a subset of the DataFrame's columns based on the column dtypes.

    Parameters
    -----
    include, exclude : scalar or list-like
        A selection of dtypes or strings to be included/excluded. At least
        one of these parameters must be supplied.

    Returns
    -----
    DataFrame
        The subset of the frame including the dtypes in ``include`` and
        excluding the dtypes in ``exclude``.

    Raises
    -----
    ValueError
        * If both of ``include`` and ``exclude`` are empty
        * If ``include`` and ``exclude`` have overlapping elements
        * If any kind of string dtype is passed in.

    Notes
    -----
    * To select all *numeric* types, use ``np.number`` or ``'number'``
    * To select strings you must use the ``object`` dtype, but note that
      this will return *all* object dtype columns
    * See the `numpy dtype hierarchy
      <https://docs.scipy.org/doc/numpy/reference/arrays.scalars.html>`__
    * To select datetimes, use ``np.datetime64``, ``'datetime'`` or
      ``'datetime64'``
    * To select timedeltas, use ``np.timedelta64``, ``'timedelta'`` or
      ``'timedelta64'``
    * To select Pandas categorical dtypes, use ``'category'``
    * To select Pandas datetimetz dtypes, use ``'datetimetz'`` (new in
      0.20.0) or ``'datetime64[ns, tz]'``

    Examples
    -----
    >>> df = pd.DataFrame({'a': [1, 2] * 3,
...                      'b': [True, False] * 3,
...                      'c': [1.0, 2.0] * 3})
    >>> df
       a      b   c
    0    1  True  1.0
    1    2  False  2.0
    2    1  True  1.0
    3    2  False  2.0
    4    1  True  1.0
    5    2  False  2.0

    >>> df.select_dtypes(include='bool')
        b
    0  True
    1 False
    2  True
    3 False
    4  True
    5 False

```

```

>>> df.select_dtypes(include=['float64'])
      c
0  1.0
1  2.0
2  1.0
3  2.0
4  1.0
5  2.0

>>> df.select_dtypes(exclude=['int'])
      b      c
0  True   1.0
1 False   2.0
2  True   1.0
3 False   2.0
4  True   1.0
5 False   2.0
"""

if not is_list_like(include):
    include = (include,) if include is not None else ()
if not is_list_like(exclude):
    exclude = (exclude,) if exclude is not None else ()

selection = (frozenset(include), frozenset(exclude))

if not any(selection):
    raise ValueError("at least one of include or exclude must be nonempty")

# convert the myriad valid dtypes object to a single representation
include = frozenset(infer_dtype_from_object(x) for x in include)
exclude = frozenset(infer_dtype_from_object(x) for x in exclude)
for dtypes in (include, exclude):
    invalidate_string_dtypes(dtypes)

# can't both include AND exclude!
if not include.isdisjoint(exclude):
    raise ValueError(f"include and exclude overlap on {(include & exclude)}")

# We raise when both include and exclude are empty
# Hence, we can just shrink the columns we want to keep
keep_these = np.full(self.shape[1], True)

def extract_unique_dtypes_from_dtypes_set(
    dtypes_set: FrozenSet[Dtype], unique_dtypes: np.ndarray
) -> List[Dtype]:
    extracted_dtypes = [
        unique_dtype
        for unique_dtype in unique_dtypes
        if issubclass(unique_dtype.type, tuple(dtypes_set)) # type: ignore
    ]
    return extracted_dtypes

unique_dtypes = self.dtypes.unique()

if include:
    included_dtypes = extract_unique_dtypes_from_dtypes_set(
        include, unique_dtypes
    )
    keep_these &= self.dtypes.isin(included_dtypes)

if exclude:
    excluded_dtypes = extract_unique_dtypes_from_dtypes_set(
        exclude, unique_dtypes

```

```
)  
    keep_these &= ~self.dtypes.isin(excluded_dtypes)  
  
    return self.iloc[:, keep_these.values]  
  
def insert(self, loc, column, value, allow_duplicates=False) -> None:  
    """  
        Insert column into DataFrame at specified location.  
  
        Raises a ValueError if `column` is already contained in the DataFrame,  
        unless `allow_duplicates` is set to True.  
  
    Parameters  
    -----  
    loc : int  
        Insertion index. Must verify 0 <= loc <= len(columns).  
    column : str, number, or hashable object  
        Label of the inserted column.  
    value : int, Series, or array-like  
    allow_duplicates : bool, optional  
    """  
    self._ensure_valid_index(value)  
    value = self._sanitize_column(column, value, broadcast=False)  
    self._mgr.insert(loc, column, value, allow_duplicates=allow_duplicates)  
  
def assign(self, **kwargs) -> "DataFrame":  
    """  
        Assign new columns to a DataFrame.  
  
        Returns a new object with all original columns in addition to new ones.  
        Existing columns that are re-assigned will be overwritten.  
  
    Parameters  
    -----  
    **kwargs : dict of {str: callable or Series}  
        The column names are keywords. If the values are  
        callable, they are computed on the DataFrame and  
        assigned to the new columns. The callable must not  
        change input DataFrame (though pandas doesn't check it).  
        If the values are not callable, (e.g. a Series, scalar, or array),  
        they are simply assigned.  
  
    Returns  
    -----  
    DataFrame  
        A new DataFrame with the new columns in addition to  
        all the existing columns.  
  
    Notes  
    ----  
    Assigning multiple columns within the same ``assign`` is possible.  
    Later items in '\*\*kwargs' may refer to newly created or modified  
    columns in 'df'; items are computed and assigned into 'df' in order.  
.. versionchanged:: 0.23.0  
    Keyword argument order is maintained.  
  
Examples  
-----  
>>> df = pd.DataFrame({'temp_c': [17.0, 25.0]},  
...                 index=['Portland', 'Berkeley'])  
>>> df
```

```
      temp_c
Portland    17.0
Berkeley   25.0
```

Where the value is a callable, evaluated on `df`:

```
>>> df.assign(temp_f=lambda x: x.temp_c * 9 / 5 + 32)
      temp_c  temp_f
Portland    17.0    62.6
Berkeley   25.0    77.0
```

Alternatively, the same behavior can be achieved by directly referencing an existing Series or sequence:

```
>>> df.assign(temp_f=df['temp_c'] * 9 / 5 + 32)
      temp_c  temp_f
Portland    17.0    62.6
Berkeley   25.0    77.0
```

You can create multiple columns within the same assign where one of the columns depends on another one defined within the same assign:

```
>>> df.assign(temp_f=lambda x: x['temp_c'] * 9 / 5 + 32,
...             temp_k=lambda x: (x['temp_f'] + 459.67) * 5 / 9)
      temp_c  temp_f  temp_k
Portland    17.0    62.6    290.15
Berkeley   25.0    77.0    298.15
"""
data = self.copy()

for k, v in kwargs.items():
    data[k] = com.apply_if_callable(v, data)
return data

def _sanitize_column(self, key, value, broadcast=True):
    """
    Ensures new columns (which go into the BlockManager as new blocks) are
    always copied and converted into an array.

    Parameters
    -----
    key : object
    value : scalar, Series, or array-like
    broadcast : bool, default True
        If ``key`` matches multiple duplicate column names in the
        DataFrame, this parameter indicates whether ``value`` should be
        tiled so that the returned array contains a (duplicated) column for
        each occurrence of the key. If False, ``value`` will not be tiled.

    Returns
    -----
    numpy.ndarray
    """

    def reindexer(value):
        # reindex if necessary

        if value.index.equals(self.index) or not len(self.index):
            value = value._values.copy()
        else:

            # GH 4107
            try:
```

```

        value = value.reindex(self.index)._values
    except ValueError as err:
        # raised in MultiIndex.from_tuples, see test_insert_error_msmgs
        if not value.index.is_unique:
            # duplicate axis
            raise err

        # other
        raise TypeError(
            "incompatible index of inserted column with frame index"
        ) from err
    return value

if isinstance(value, Series):
    value = reindexer(value)

elif isinstance(value, DataFrame):
    # align right-hand-side columns if self.columns
    # is multi-index and self[key] is a sub-frame
    if isinstance(self.columns, ABCMultiIndex) and key in self.columns:
        loc = self.columns.get_loc(key)
        if isinstance(loc, (slice, Series, np.ndarray, Index)):
            cols = maybe_droplevels(self.columns[loc], key)
            if len(cols) and not cols.equals(value.columns):
                value = value.reindex(cols, axis=1)
    # now align rows
    value = reindexer(value).T

elif isinstance(value, ExtensionArray):
    # Explicitly copy here, instead of in sanitize_index,
    # as sanitize_index won't copy an EA, even with copy=True
    value = value.copy()
    value = sanitize_index(value, self.index)

elif isinstance(value, Index) or is_sequence(value):

    # turn me into an ndarray
    value = sanitize_index(value, self.index)
    if not isinstance(value, (np.ndarray, Index)):
        if isinstance(value, list) and len(value) > 0:
            value = maybe_convert_platform(value)
        else:
            value = com.asarray_tuplesafe(value)
    elif value.ndim == 2:
        value = value.copy().T
    elif isinstance(value, Index):
        value = value.copy(deep=True)
    else:
        value = value.copy()

    # possibly infer to datetimelike
    if is_object_dtype(value.dtype):
        value = maybe_infer_to_datetimelike(value)

else:
    # cast ignores pandas dtypes. so save the dtype first
    infer_dtype, _ = infer_dtype_from_scalar(value, pandas_dtype=True)

    # upcast
    value = cast_scalar_to_array(len(self.index), value)
    value = maybe_cast_to_datetime(value, infer_dtype)

# return internal types directly

```

```

if is_extension_array_dtype(value):
    return value

# broadcast across multiple columns if necessary
if broadcast and key in self.columns and value.ndim == 1:
    if not self.columns.is_unique or isinstance(self.columns, ABCMultiIndex):
        existing_piece = self[key]
        if isinstance(existing_piece, DataFrame):
            value = np.tile(value, (len(existing_piece.columns), 1))

return np.atleast_2d(np.asarray(value))

@property
def _series(self):
    return {
        item: Series(
            self._mgr.iget(idx), index=self.index, name=item, fastpath=True
        )
        for idx, item in enumerate(self.columns)
    }

def lookup(self, row_labels, col_labels) -> np.ndarray:
    """
    Label-based "fancy indexing" function for DataFrame.

    Given equal-length arrays of row and column labels, return an array of the values corresponding to each (row, col) pair.

    Parameters
    -----
    row_labels : sequence
        The row labels to use for lookup.
    col_labels : sequence
        The column labels to use for lookup.

    Returns
    -----
    numpy.ndarray
        The found values.
    """
    n = len(row_labels)
    if n != len(col_labels):
        raise ValueError("Row labels must have same size as column labels")
    if not (self.index.is_unique and self.columns.is_unique):
        # GH#33041
        raise ValueError("DataFrame.lookup requires unique index and columns")

    thresh = 1000
    if not self._is_mixed_type or n > thresh:
        values = self.values
        ridx = self.index.get_indexer(row_labels)
        cidx = self.columns.get_indexer(col_labels)
        if (ridx == -1).any():
            raise KeyError("One or more row labels was not found")
        if (cidx == -1).any():
            raise KeyError("One or more column labels was not found")
        flat_index = ridx * len(self.columns) + cidx
        result = values.flat[flat_index]
    else:
        result = np.empty(n, dtype="O")
        for i, (r, c) in enumerate(zip(row_labels, col_labels)):
            result[i] = self._get_value(r, c)

```

```
    if is_object_dtype(result):
        result = lib.maybe_convert_objects(result)

    return result

# -----
# Reindexing and alignment

def _reindex_axes(self, axes, level, tolerance, method, fill_value, copy):
    frame = self

    columns = axes["columns"]
    if columns is not None:
        frame = frame._reindex_columns(
            columns, method, copy, level, fill_value, limit, tolerance
        )

    index = axes["index"]
    if index is not None:
        frame = frame._reindex_index(
            index, method, copy, level, fill_value, limit, tolerance
        )

    return frame

def _reindex_index(
    self,
    new_index,
    method,
    copy,
    level,
    fill_value=np.nan,
    limit=None,
    tolerance=None,
):
    new_index, indexer = self.index.reindex(
        new_index, method=method, level=level, limit=limit, tolerance=tolerance
    )
    return self._reindex_with_indexers(
        {0: [new_index, indexer]},
        copy=copy,
        fill_value=fill_value,
        allow_dups=False,
    )

def _reindex_columns(
    self,
    new_columns,
    method,
    copy,
    level,
    fill_value=None,
    limit=None,
    tolerance=None,
):
    new_columns, indexer = self.columns.reindex(
        new_columns, method=method, level=level, limit=limit, tolerance=tolerance
    )
    return self._reindex_with_indexers(
        {1: [new_columns, indexer]},
        copy=copy,
        fill_value=fill_value,
        allow_dups=False,
```

```

    )

def _reindex_multi(self, axes, copy, fill_value) -> "DataFrame":
    """
    We are guaranteed non-Nones in the axes.
    """
    new_index, row_indexer = self.index.reindex(axes["index"])
    new_columns, col_indexer = self.columns.reindex(axes["columns"])

    if row_indexer is not None and col_indexer is not None:
        indexer = row_indexer, col_indexer
        new_values = algorithms.take_2d_multi(
            self.values, indexer, fill_value=fill_value
        )
    else:
        return self._constructor(new_values, index=new_index, columns=new_columns)

    return self._reindex_with_indexers(
        {0: [new_index, row_indexer], 1: [new_columns, col_indexer]},
        copy=copy,
        fill_value=fill_value,
    )

@doc(NDFrame.align, **_shared_doc_kwargs)
def align(
    self,
    other,
    join="outer",
    axis=None,
    level=None,
    copy=True,
    fill_value=None,
    method=None,
    limit=None,
    fill_axis=0,
    broadcast_axis=None,
) -> "DataFrame":
    return super().align(
        other,
        join=join,
        axis=axis,
        level=level,
        copy=copy,
        fill_value=fill_value,
        method=method,
        limit=limit,
        fill_axis=fill_axis,
        broadcast_axis=broadcast_axis,
    )

@Appender(
    """
    Examples
    -----
    >>> df = pd.DataFrame({ "A": [1, 2, 3], "B": [4, 5, 6] })

    Change the row labels.

    >>> df.set_axis(['a', 'b', 'c'], axis='index')
      A   B
    a   1   4
    b   2   5
    c   3   6
    """)

```

```
Change the column labels.
```

```
>>> df.set_axis(['I', 'II'], axis='columns')
      I   II
0    1    4
1    2    5
2    3    6

Now, update the labels inplace.

>>> df.set_axis(['i', 'ii'], axis='columns', inplace=True)
>>> df
      i   ii
0    1    4
1    2    5
2    3    6
"""

)
@Substitution(
    **_shared_doc_kwargs,
    extended_summary_sub=" column or",
    axis_description_sub=", and 1 identifies the columns",
    see_also_sub=" or columns",
)
@Appender(NDFrame.set_axis.__doc__)
def set_axis(self, labels, axis: Axis = 0, inplace: bool = False):
    return super().set_axis(labels, axis=axis, inplace=inplace)

@Substitution(**_shared_doc_kwargs)
@Appender(NDFrame.reindex.__doc__)
@rewrite_axis_style_signature(
    "labels",
    [
        ("method", None),
        ("copy", True),
        ("level", None),
        ("fill_value", np.nan),
        ("limit", None),
        ("tolerance", None),
    ],
)
def reindex(self, *args, **kwargs) -> "DataFrame":
    axes = validate_axis_style_args(self, args, kwargs, "labels", "reindex")
    kwargs.update(axes)
    # Pop these, since the values are in `kwargs` under different names
    kwargs.pop("axis", None)
    kwargs.pop("labels", None)
    return super().reindex(**kwargs)

def drop(
    self,
    labels=None,
    axis=0,
    index=None,
    columns=None,
    level=None,
    inplace=False,
    errors="raise",
):
    """
    Drop specified labels from rows or columns.

    Remove rows or columns by specifying label names and corresponding

```

axis, or by specifying directly index or column names. When using a multi-index, labels on different levels can be removed by specifying the level.

Parameters

labels : single label or list-like
Index or column labels to drop.
axis : {0 or 'index', 1 or 'columns'}, default 0
Whether to drop labels from the index (0 or 'index') or columns (1 or 'columns').
index : single label or list-like
Alternative to specifying axis (``labels, axis=0`` is equivalent to ``index=labels``).
columns : single label or list-like
Alternative to specifying axis (``labels, axis=1`` is equivalent to ``columns=labels``).
level : int or level name, optional
For MultiIndex, level from which the labels will be removed.
inplace : bool, default False
If True, do operation inplace and return None.
errors : {'ignore', 'raise'}, default 'raise'
If 'ignore', suppress error and only existing labels are dropped.

Returns

DataFrame
DataFrame without the removed index or column labels.

Raises

KeyError
If any of the labels is not found in the selected axis.

See Also

DataFrame.loc : Label-location based indexer for selection by label.
DataFrame.dropna : Return DataFrame with labels on given axis omitted where (all or any) data are missing.
DataFrame.drop_duplicates : Return DataFrame with duplicate rows removed, optionally only considering certain columns.
Series.drop : Return Series with specified index labels removed.

Examples

```
>>> df = pd.DataFrame(np.arange(12).reshape(3, 4),  
...                      columns=['A', 'B', 'C', 'D'])  
>>> df  
   A   B   C   D  
0  0   1   2   3  
1  4   5   6   7  
2  8   9  10  11
```

Drop columns

```
>>> df.drop(['B', 'C'], axis=1)
```

```
   A   D  
0  0   3  
1  4   7  
2  8  11
```

```
>>> df.drop(columns=['B', 'C'])
```

```

      A    D
0    0    3
1    4    7
2    8   11

Drop a row by index

>>> df.drop([0, 1])
      A    B    C    D
2    8    9   10   11

Drop columns and/or rows of MultiIndex DataFrame

>>> midx = pd.MultiIndex(levels=[['lama', 'cow', 'falcon'],
...                               ['speed', 'weight', 'length']],
...                           codes=[[0, 0, 0, 1, 1, 1, 2, 2, 2],
...                                  [0, 1, 2, 0, 1, 2, 0, 1, 2]])
>>> df = pd.DataFrame(index=midx, columns=['big', 'small'],
...                      data=[[45, 30], [200, 100], [1.5, 1], [30, 20],
...                             [250, 150], [1.5, 0.8], [320, 250],
...                             [1, 0.8], [0.3, 0.2]])
>>> df
              big      small
lama    speed    45.0     30.0
        weight   200.0    100.0
        length    1.5     1.0
cow     speed    30.0     20.0
        weight   250.0    150.0
        length    1.5     0.8
falcon  speed   320.0    250.0
        weight    1.0     0.8
        length    0.3     0.2

>>> df.drop(index='cow', columns='small')
              big
lama    speed    45.0
        weight   200.0
        length    1.5
falcon  speed   320.0
        weight    1.0
        length    0.3

>>> df.drop(index='length', level=1)
              big      small
lama    speed    45.0     30.0
        weight   200.0    100.0
cow     speed    30.0     20.0
        weight   250.0    150.0
falcon  speed   320.0    250.0
        weight    1.0     0.8
"""

return super().drop(
    labels=labels,
    axis=axis,
    index=index,
    columns=columns,
    level=level,
    inplace=inplace,
    errors=errors,
)

@rewrite_axis_style_signature(
    "mapper",

```

```
[("copy", True), ("inplace", False), ("level", None), ("errors", "ignore")],  
)  
def rename(  
    self,  
    mapper: Optional[Renamer] = None,  
    *,  
    index: Optional[Renamer] = None,  
    columns: Optional[Renamer] = None,  
    axis: Optional[Axis] = None,  
    copy: bool = True,  
    inplace: bool = False,  
    level: Optional[Level] = None,  
    errors: str = "ignore",  
) -> Optional["DataFrame"]:  
    """  
        Alter axes labels.  
  
        Function / dict values must be unique (1-to-1). Labels not contained in  
        a dict / Series will be left as-is. Extra labels listed don't throw an  
        error.  
  
        See the :ref:`user guide <basics.rename>` for more.  
  
    Parameters  
    -----  
    mapper : dict-like or function  
        Dict-like or functions transformations to apply to  
        that axis' values. Use either ``mapper`` and ``axis`` to  
        specify the axis to target with ``mapper``, or ``index`` and  
        ``columns``.  
    index : dict-like or function  
        Alternative to specifying axis (``mapper, axis=0``  
        is equivalent to ``index=mapper``).  
    columns : dict-like or function  
        Alternative to specifying axis (``mapper, axis=1``  
        is equivalent to ``columns=mapper``).  
    axis : {0 or 'index', 1 or 'columns'}, default 0  
        Axis to target with ``mapper``. Can be either the axis name  
        ('index', 'columns') or number (0, 1). The default is 'index'.  
    copy : bool, default True  
        Also copy underlying data.  
    inplace : bool, default False  
        Whether to return a new DataFrame. If True then value of copy is  
        ignored.  
    level : int or level name, default None  
        In case of a MultiIndex, only rename labels in the specified  
        level.  
    errors : {'ignore', 'raise'}, default 'ignore'  
        If 'raise', raise a `KeyError` when a dict-like `mapper`, `index`,  
        or `columns` contains labels that are not present in the Index  
        being transformed.  
        If 'ignore', existing keys will be renamed and extra keys will be  
        ignored.  
  
    Returns  
    -----  
    DataFrame  
        DataFrame with the renamed axis labels.  
  
    Raises  
    -----  
    KeyError  
        If any of the labels is not found in the selected axis and
```

```
"errors='raise'".

See Also
-----
DataFrame.rename_axis : Set the name of the axis.

Examples
-----
```
DataFrame.rename`` supports two calling conventions

* ``(index=index_mapper, columns=columns_mapper, ...)``

* ``(mapper, axis={'index', 'columns'}, ...)``

We *highly* recommend using keyword arguments to clarify your intent.
```

Rename columns using a mapping:

```
>>> df = pd.DataFrame({"A": [1, 2, 3], "B": [4, 5, 6]})
>>> df.rename(columns={"A": "a", "B": "c"})
 a c
0 1 4
1 2 5
2 3 6
```

Rename index using a mapping:

```
>>> df.rename(index={0: "x", 1: "y", 2: "z"})
 A B
x 1 4
y 2 5
z 3 6
```

Cast index labels to a different type:

```
>>> df.index
RangeIndex(start=0, stop=3, step=1)
>>> df.rename(index=str).index
Index(['0', '1', '2'], dtype='object')

>>> df.rename(columns={"A": "a", "B": "b", "C": "c"}, errors="raise")
Traceback (most recent call last):
KeyError: ['C'] not found in axis
```

Using axis-style parameters

```
>>> df.rename(str.lower, axis='columns')
 a b
0 1 4
1 2 5
2 3 6

>>> df.rename({1: 2, 2: 4}, axis='index')
 A B
0 1 4
2 2 5
4 3 6
"""
return super().rename(
 mapper=mapper,
 index=index,
 columns=columns,
 axis=axis,
```

```
 copy=copy,
 inplace=inplace,
 level=level,
 errors=errors,
)

@doc(NDFrame.fillna, **_shared_doc_kwargs)
def fillna(
 self,
 value=None,
 method=None,
 axis=None,
 inplace=False,
 limit=None,
 downcast=None,
) -> Optional["DataFrame"]:
 return super().fillna(
 value=value,
 method=method,
 axis=axis,
 inplace=inplace,
 limit=limit,
 downcast=downcast,
)

@doc(NDFrame.replace, **_shared_doc_kwargs)
def replace(
 self,
 to_replace=None,
 value=None,
 inplace=False,
 limit=None,
 regex=False,
 method="pad",
):
 return super().replace(
 to_replace=to_replace,
 value=value,
 inplace=inplace,
 limit=limit,
 regex=regex,
 method=method,
)

def _replace_columnwise(
 self, mapping: Dict[Label, Tuple[Any, Any]], inplace: bool, regex
):
 """
 Dispatch to Series.replace column-wise.

 Parameters

 mapping : dict
 of the form {col: (target, value)}
 inplace : bool
 regex : bool or same types as `to_replace` in DataFrame.replace

 Returns

 DataFrame or None
 """
 # Operate column-wise
```

```

res = self if inplace else self.copy()
ax = self.columns

for i in range(len(ax)):
 if ax[i] in mapping:
 ser = self.iloc[:, i]

 target, value = mapping[ax[i]]
 newobj = ser.replace(target, value, regex=regex)

 res.iloc[:, i] = newobj

if inplace:
 return
return res.__finalize__(self)

@doc(NDFrame.shift, klass=_shared_doc_kwargs["klass"])
def shift(self, periods=1, freq=None, axis=0, fill_value=None) -> "DataFrame":
 return super().shift(
 periods=periods, freq=freq, axis=axis, fill_value=fill_value
)

def set_index(
 self, keys, drop=True, append=False, inplace=False, verify_integrity=False
):
 """
 Set the DataFrame index using existing columns.

 Set the DataFrame index (row labels) using one or more existing
 columns or arrays (of the correct length). The index can replace the
 existing index or expand on it.

 Parameters

 keys : label or array-like or list of labels/arrays
 This parameter can be either a single column key, a single array of
 the same length as the calling DataFrame, or a list containing an
 arbitrary combination of column keys and arrays. Here, "array"
 encompasses :class:`Series`, :class:`Index`, ``np.ndarray``, and
 instances of :class:`~collections.abc.Iterator`.
 drop : bool, default True
 Delete columns to be used as the new index.
 append : bool, default False
 Whether to append columns to existing index.
 inplace : bool, default False
 Modify the DataFrame in place (do not create a new object).
 verify_integrity : bool, default False
 Check the new index for duplicates. Otherwise defer the check until
 necessary. Setting to False will improve the performance of this
 method.

 Returns

 DataFrame
 Changed row labels.

 See Also

 DataFrame.reset_index : Opposite of set_index.
 DataFrame.reindex : Change to new indices or expand indices.
 DataFrame.reindex_like : Change to same indices as other DataFrame.
 Examples

```

```

>>> df = pd.DataFrame({'month': [1, 4, 7, 10],
... 'year': [2012, 2014, 2013, 2014],
... 'sale': [55, 40, 84, 31]})
>>> df
 month year sale
0 1 2012 55
1 4 2014 40
2 7 2013 84
3 10 2014 31

Set the index to become the 'month' column:

>>> df.set_index('month')
 year sale
month
1 2012 55
4 2014 40
7 2013 84
10 2014 31

Create a MultiIndex using columns 'year' and 'month':

>>> df.set_index(['year', 'month'])
 sale
year month
2012 1 55
2014 4 40
2013 7 84
2014 10 31

Create a MultiIndex using an Index and a column:

>>> df.set_index([pd.Index([1, 2, 3, 4]), 'year'])
 month sale
year
1 2012 1 55
2 2014 4 40
3 2013 7 84
4 2014 10 31

Create a MultiIndex using two Series:

>>> s = pd.Series([1, 2, 3, 4])
>>> df.set_index([s, s**2])
 month year sale
1 1 1 2012 55
2 4 4 2014 40
3 9 7 2013 84
4 16 10 2014 31
"""

inplace = validate_bool_kwarg(inplace, "inplace")
if not isinstance(keys, list):
 keys = [keys]

err_msg = (
 'The parameter "keys" may be a column key, one-dimensional '
 'array, or a list containing only valid column keys and '
 'one-dimensional arrays.'
)
missing: List[Label] = []
for col in keys:

```

```

 if isinstance(
 col, (ABCIndexClass, ABCSeries, np.ndarray, list, abc.Iterator)
):
 # arrays are fine as long as they are one-dimensional
 # iterators get converted to list below
 if getattr(col, "ndim", 1) != 1:
 raise ValueError(err_msg)
 else:
 # everything else gets tried as a key; see GH 24969
 try:
 found = col in self.columns
 except TypeError as err:
 raise TypeError(
 f"{err_msg}. Received column of type {type(col)}"
) from err
 else:
 if not found:
 missing.append(col)

 if missing:
 raise KeyError(f"None of {missing} are in the columns")

 if inplace:
 frame = self
 else:
 frame = self.copy()

 arrays = []
 names = []
 if append:
 names = list(self.index.names)
 if isinstance(self.index, ABCMultiIndex):
 for i in range(self.index.nlevels):
 arrays.append(self.index._get_level_values(i))
 else:
 arrays.append(self.index)

 to_remove: List[Label] = []
 for col in keys:
 if isinstance(col, ABCMultiIndex):
 for n in range(col.nlevels):
 arrays.append(col._get_level_values(n))
 names.extend(col.names)
 elif isinstance(col, (ABCIndexClass, ABCSeries)):
 # if Index then not MultiIndex (treated above)
 arrays.append(col)
 names.append(col.name)
 elif isinstance(col, (list, np.ndarray)):
 arrays.append(col)
 names.append(None)
 elif isinstance(col, abc.Iterator):
 arrays.append(list(col))
 names.append(None)
 # from here, col can only be a column label
 else:
 arrays.append(frame[col]._values)
 names.append(col)
 if drop:
 to_remove.append(col)

 if len(arrays[-1]) != len(self):
 # check newest element against length of calling frame, since
 # ensure_index from sequences would not raise for append=False.

```

```

 raise ValueError(
 f"Length mismatch: Expected {len(self)} rows, "
 f"received array of length {len(arrays[-1])}"
)

index = ensure_index_from_sequences(arrays, names)

if verify_integrity and not index.is_unique:
 duplicates = index[index.duplicated()].unique()
 raise ValueError(f"Index has duplicate keys: {duplicates}")

use set to handle duplicate column names gracefully in case of drop
for c in set(to_remove):
 del frame[c]

clear up memory usage
index._cleanup()

frame.index = index

if not inplace:
 return frame

def reset_index(
 self,
 level: Optional[Union[Hashable, Sequence[Hashable]]] = None,
 drop: bool = False,
 inplace: bool = False,
 col_level: Hashable = 0,
 col_fill: Label = "",
) -> Optional["DataFrame"]:
 """
 Reset the index, or a level of it.

 Reset the index of the DataFrame, and use the default one instead.
 If the DataFrame has a MultiIndex, this method can remove one or more
 levels.

 Parameters

 level : int, str, tuple, or list, default None
 Only remove the given levels from the index. Removes all levels by
 default.
 drop : bool, default False
 Do not try to insert index into dataframe columns. This resets
 the index to the default integer index.
 inplace : bool, default False
 Modify the DataFrame in place (do not create a new object).
 col_level : int or str, default 0
 If the columns have multiple levels, determines which level the
 labels are inserted into. By default it is inserted into the first
 level.
 col_fill : object, default ''
 If the columns have multiple levels, determines how the other
 levels are named. If None then the index name is repeated.

 Returns

 DataFrame or None
 DataFrame with the new index or None if ``inplace=True``.

 See Also

```

```
DataFrame.set_index : Opposite of reset_index.
DataFrame.reindex : Change to new indices or expand indices.
DataFrame.reindex_like : Change to same indices as other DataFrame.
```

#### Examples

-----

```
>>> df = pd.DataFrame([('bird', 389.0),
... ('bird', 24.0),
... ('mammal', 80.5),
... ('mammal', np.nan)],
... index=['falcon', 'parrot', 'lion', 'monkey'],
... columns=('class', 'max_speed'))
>>> df
 class max_speed
falcon bird 389.0
parrot bird 24.0
lion mammal 80.5
monkey mammal NaN
```

When we reset the index, the old index is added as a column, and a new sequential index is used:

```
>>> df.reset_index()
 index class max_speed
0 falcon bird 389.0
1 parrot bird 24.0
2 lion mammal 80.5
3 monkey mammal NaN
```

We can use the `drop` parameter to avoid the old index being added as a column:

```
>>> df.reset_index(drop=True)
 class max_speed
0 bird 389.0
1 bird 24.0
2 mammal 80.5
3 mammal NaN
```

You can also use `reset\_index` with `MultiIndex`.

```
>>> index = pd.MultiIndex.from_tuples([('bird', 'falcon'),
... ('bird', 'parrot'),
... ('mammal', 'lion'),
... ('mammal', 'monkey')],
... names=['class', 'name'])
>>> columns = pd.MultiIndex.from_tuples([('speed', 'max'),
... ('species', 'type')])
>>> df = pd.DataFrame([(389.0, 'fly'),
... (24.0, 'fly'),
... (80.5, 'run'),
... (np.nan, 'jump')],
... index=index,
... columns=columns)
>>> df
 speed species
 max type
class name
bird falcon 389.0 fly
 parrot 24.0 fly
mammal lion 80.5 run
 monkey NaN jump
```

```
If the index has multiple levels, we can reset a subset of them:
```

```
>>> df.reset_index(level='class')
 class speed species
 max type
name
falcon bird 389.0 fly
parrot bird 24.0 fly
lion mammal 80.5 run
monkey mammal NaN jump
```

If we are not dropping the index, by default, it is placed in the top level. We can place it in another level:

```
>>> df.reset_index(level='class', col_level=1)
 speed species
 class max type
name
falcon bird 389.0 fly
parrot bird 24.0 fly
lion mammal 80.5 run
monkey mammal NaN jump
```

When the index is inserted under another level, we can specify under which one with the parameter `col\_fill`:

```
>>> df.reset_index(level='class', col_level=1, col_fill='species')
 species speed species
 class max type
name
falcon bird 389.0 fly
parrot bird 24.0 fly
lion mammal 80.5 run
monkey mammal NaN jump
```

If we specify a nonexistent level for `col\_fill`, it is created:

```
>>> df.reset_index(level='class', col_level=1, col_fill='genus')
 genus speed species
 class max type
name
falcon bird 389.0 fly
parrot bird 24.0 fly
lion mammal 80.5 run
monkey mammal NaN jump
"""
inplace = validate_bool_kwarg(inplace, "inplace")
if inplace:
 new_obj = self
else:
 new_obj = self.copy()

def _maybe_casted_values(index, labels=None):
 values = index._values
 if not isinstance(index, (PeriodIndex, DatetimeIndex)):
 if values.dtype == np.object_:
 values = lib.maybe_convert_objects(values)

 # if we have the labels, extract the values with a mask
 if labels is not None:
 mask = labels == -1

 # we can have situations where the whole mask is -1,
```

```

meaning there is nothing found in labels, so make all nan's
if mask.all():
 values = np.empty(len(mask))
 values.fill(np.nan)
else:
 values = values.take(labels)

TODO(https://github.com/pandas-dev/pandas/issues/24206)
Push this into maybe_upcast_putmask?
We can't pass EAs there right now. Looks a bit
complicated.
So we unbox the ndarray_values, op, re-box.
values_type = type(values)
values_dtype = values.dtype

if issubclass(values_type, DatetimeLikeArray):
 values = values._data # TODO: can we de-kludge yet?

if mask.any():
 values, _ = maybe_upcast_putmask(values, mask, np.nan)

if issubclass(values_type, DatetimeLikeArray):
 values = values_type(values, dtype=values_dtype)

return values

new_index = ibase.default_index(len(new_obj))
if level is not None:
 if not isinstance(level, (tuple, list)):
 level = [level]
 level = [self.index._get_level_number(lev) for lev in level]
 if len(level) < self.index.nlevels:
 new_index = self.index.droplevel(level)

if not drop:
 to_insert: Iterable[Tuple[Any, Optional[Any]]]
 if isinstance(self.index, ABCMultiIndex):
 names = [
 (n if n is not None else f"level_{i}")
 for i, n in enumerate(self.index.names)
]
 to_insert = zip(self.index.levels, self.index.codes)
 else:
 default = "index" if "index" not in self else "level_0"
 names = [default] if self.index.name is None else [self.index.name]
 to_insert = ((self.index, None),)

multi_col = isinstance(self.columns, ABCMultiIndex)
for i, (lev, lab) in reversed(list(enumerate(to_insert))):
 if not (level is None or i in level):
 continue
 name = names[i]
 if multi_col:
 col_name = list(name) if isinstance(name, tuple) else [name]
 if col_fill is None:
 if len(col_name) not in (1, self.columns.nlevels):
 raise ValueError(
 "col_fill=None is incompatible "
 f"with incomplete column name {name}"
)
 col_fill = col_name[0]

 lev_num = self.columns._get_level_number(col_level)

```

```

 name_lst = [col_fill] * lev_num + col_name
 missing = self.columns.nlevels - len(name_lst)
 name_lst += [col_fill] * missing
 name = tuple(name_lst)
 # to ndarray and maybe infer different dtype
 level_values = _maybe_casted_values(lev, lab)
 new_obj.insert(0, name, level_values)

 new_obj.index = new_index
 if not inplace:
 return new_obj

 return None

Reindex-based selection methods

@Appender(_shared_docs["isna"] % _shared_doc_kwargs)
def isna(self) -> "DataFrame":
 result = self._constructor(self._data.isna(func=isna))
 return result._finalize_(self, method="isna")

@Appender(_shared_docs["isna"] % _shared_doc_kwargs)
def isnull(self) -> "DataFrame":
 return self.isna()

@Appender(_shared_docs["notna"] % _shared_doc_kwargs)
def notna(self) -> "DataFrame":
 return ~self.isna()

@Appender(_shared_docs["notna"] % _shared_doc_kwargs)
def notnull(self) -> "DataFrame":
 return ~self.isna()

def dropna(self, axis=0, how="any", thresh=None, subset=None, inplace=False):
 """
 Remove missing values.

 See the :ref:`User Guide <missing_data>` for more on which values are
 considered missing, and how to work with missing data.

 Parameters

 axis : {0 or 'index', 1 or 'columns'}, default 0
 Determine if rows or columns which contain missing values are
 removed.

 * 0, or 'index' : Drop rows which contain missing values.
 * 1, or 'columns' : Drop columns which contain missing value.

 .. versionchanged:: 1.0.0

 Pass tuple or list to drop on multiple axes.
 Only a single axis is allowed.

 how : {'any', 'all'}, default 'any'
 Determine if row or column is removed from DataFrame, when we have
 at least one NA or all NA.

 * 'any' : If any NA values are present, drop that row or column.
 * 'all' : If all values are NA, drop that row or column.

 thresh : int, optional

```

```
 Require that many non-NA values.
subset : array-like, optional
 Labels along other axis to consider, e.g. if you are dropping rows
 these would be a list of columns to include.
inplace : bool, default False
 If True, do operation inplace and return None.
```

Returns

-----

DataFrame

DataFrame with NA entries dropped from it.

See Also

-----

DataFrame.isna : Indicate missing values.

DataFrame.notna : Indicate existing (non-missing) values.

DataFrame.fillna : Replace missing values.

Series.dropna : Drop missing values.

Index.dropna : Drop missing indices.

Examples

-----

```
>>> df = pd.DataFrame({"name": ['Alfred', 'Batman', 'Catwoman'],
... "toy": [np.nan, 'Batmobile', 'Bullwhip'],
... "born": [pd.NaT, pd.Timestamp("1940-04-25"),
... pd.NaT] })
>>> df
 name toy born
0 Alfred NaN NaT
1 Batman Batmobile 1940-04-25
2 Catwoman Bullwhip NaT
```

Drop the rows where at least one element is missing.

```
>>> df.dropna()
 name toy born
1 Batman Batmobile 1940-04-25
```

Drop the columns where at least one element is missing.

```
>>> df.dropna(axis='columns')
 name
0 Alfred
1 Batman
2 Catwoman
```

Drop the rows where all elements are missing.

```
>>> df.dropna(how='all')
 name toy born
0 Alfred NaN NaT
1 Batman Batmobile 1940-04-25
2 Catwoman Bullwhip NaT
```

Keep only the rows with at least 2 non-NA values.

```
>>> df.dropna(thresh=2)
 name toy born
1 Batman Batmobile 1940-04-25
2 Catwoman Bullwhip NaT
```

Define in which columns to look for missing values.

```

>>> df.dropna(subset=['name', 'born'])
 name toy born
1 Batman Batmobile 1940-04-25

Keep the DataFrame with valid entries in the same variable.

>>> df.dropna(inplace=True)
>>> df
 name toy born
1 Batman Batmobile 1940-04-25
"""

inplace = validate_bool_kwarg(inplace, "inplace")
if isinstance(axis, (tuple, list)):
 # GH20987
 raise TypeError("supplying multiple axes to axis is no longer supported.")

axis = self._get_axis_number(axis)
agg_axis = 1 - axis

agg_obj = self
if subset is not None:
 ax = self._get_axis(agg_axis)
 indices = ax.get_indexer_for(subset)
 check = indices == -1
 if check.any():
 raise KeyError(list(np.compress(check, subset)))
 agg_obj = self.take(indices, axis=agg_axis)

count = agg_obj.count(axis=agg_axis)

if thresh is not None:
 mask = count >= thresh
elif how == "any":
 mask = count == len(agg_obj._get_axis(agg_axis))
elif how == "all":
 mask = count > 0
else:
 if how is not None:
 raise ValueError(f"invalid how option: {how}")
 else:
 raise TypeError("must specify how or thresh")

result = self.loc(axis=axis) [mask]

if inplace:
 self._update_inplace(result)
else:
 return result

def drop_duplicates(
 self,
 subset: Optional[Union[Hashable, Sequence[Hashable]]] = None,
 keep: Union[str, bool] = "first",
 inplace: bool = False,
 ignore_index: bool = False,
) -> Optional["DataFrame"]:
"""
Return DataFrame with duplicate rows removed.

Considering certain columns is optional. Indexes, including time indexes
are ignored.

```

#### Parameters

```

subset : column label or sequence of labels, optional
 Only consider certain columns for identifying duplicates, by
 default use all of the columns.
keep : {'first', 'last', False}, default 'first'
 Determines which duplicates (if any) to keep.
 - ``first`` : Drop duplicates except for the first occurrence.
 - ``last`` : Drop duplicates except for the last occurrence.
 - False : Drop all duplicates.
inplace : bool, default False
 Whether to drop duplicates in place or to return a copy.
ignore_index : bool, default False
 If True, the resulting axis will be labeled 0, 1, ..., n - 1.
 .. versionadded:: 1.0.0
```

Returns

-----  
DataFrame

DataFrame with duplicates removed or None if ``inplace=True``.

See Also

-----  
DataFrame.value\_counts: Count unique combinations of columns.

Examples

-----  
Consider dataset containing ramen rating.

```
>>> df = pd.DataFrame({
... 'brand': ['Yum Yum', 'Yum Yum', 'Indomie', 'Indomie', 'Indomie'],
... 'style': ['cup', 'cup', 'cup', 'pack', 'pack'],
... 'rating': [4, 4, 3.5, 15, 5]
... })
>>> df
 brand style rating
0 Yum Yum cup 4.0
1 Yum Yum cup 4.0
2 Indomie cup 3.5
3 Indomie pack 15.0
4 Indomie pack 5.0
```

By default, it removes duplicate rows based on all columns.

```
>>> df.drop_duplicates()
 brand style rating
0 Yum Yum cup 4.0
2 Indomie cup 3.5
3 Indomie pack 15.0
4 Indomie pack 5.0
```

To remove duplicates on specific column(s), use ``subset``.

```
>>> df.drop_duplicates(subset=['brand'])
 brand style rating
0 Yum Yum cup 4.0
2 Indomie cup 3.5
```

To remove duplicates and keep last occurrences, use ``keep``.

```
>>> df.drop_duplicates(subset=['brand', 'style'], keep='last')
 brand style rating
1 Yum Yum cup 4.0
```

```

2 Indomie cup 3.5
4 Indomie pack 5.0
"""
if self.empty:
 return self.copy()

inplace = validate_bool_kwarg(inplace, "inplace")
duplicated = self.duplicated(subset, keep=keep)

result = self[-duplicated]
if ignore_index:
 result.index = ibase.default_index(len(result))

if inplace:
 self._update_inplace(result)
 return None
else:
 return result

def duplicated(
 self,
 subset: Optional[Union[Hashable, Sequence[Hashable]]] = None,
 keep: Union[str, bool] = "first",
) -> "Series":
"""
Return boolean Series denoting duplicate rows.

Considering certain columns is optional.

Parameters

subset : column label or sequence of labels, optional
 Only consider certain columns for identifying duplicates, by
 default use all of the columns.
keep : {'first', 'last', False}, default 'first'
 Determines which duplicates (if any) to mark.

 - ``first`` : Mark duplicates as ``True`` except for the first occurrence.
 - ``last`` : Mark duplicates as ``True`` except for the last occurrence.
 - False : Mark all duplicates as ``True``.

Returns

Series
 Boolean series for each duplicated rows.

See Also

Index.duplicated : Equivalent method on index.
Series.duplicated : Equivalent method on Series.
Series.drop_duplicates : Remove duplicate values from Series.
DataFrame.drop_duplicates : Remove duplicate values from DataFrame.

Examples

Consider dataset containing ramen rating.

>>> df = pd.DataFrame({
... 'brand': ['Yum Yum', 'Yum Yum', 'Indomie', 'Indomie', 'Indomie'],
... 'style': ['cup', 'cup', 'cup', 'pack', 'pack'],
... 'rating': [4, 4, 3.5, 15, 5]
... })
>>> df

```

```
 brand style rating
0 Yum Yum cup 4.0
1 Yum Yum cup 4.0
2 Indomie cup 3.5
3 Indomie pack 15.0
4 Indomie pack 5.0
```

By default, for each set of duplicated values, the first occurrence is set on False and all others on True.

```
>>> df.duplicated()
0 False
1 True
2 False
3 False
4 False
dtype: bool
```

By using 'last', the last occurrence of each set of duplicated values is set on False and all others on True.

```
>>> df.duplicated(keep='last')
0 True
1 False
2 False
3 False
4 False
dtype: bool
```

By setting ``keep`` on False, all duplicates are True.

```
>>> df.duplicated(keep=False)
0 True
1 True
2 False
3 False
4 False
dtype: bool
```

To find duplicates on specific column(s), use ``subset``.

```
>>> df.duplicated(subset=['brand'])
0 False
1 True
2 False
3 True
4 True
dtype: bool
"""
from pandas.core.sorting import get_group_index
from pandas._libs.hashtable import duplicated_int64, _SIZE_HINT_LIMIT

if self.empty:
 return Series(dtype=bool)

def f(vals):
 labels, shape = algorithms.factorize(
 vals, size_hint=min(len(self), _SIZE_HINT_LIMIT)
)
 return labels.astype("i8", copy=False), len(shape)

if subset is None:
 subset = self.columns
```

```

 elif (
 not np.iterable(subset)
 or isinstance(subset, str)
 or isinstance(subset, tuple)
 and subset in self.columns
):
 subset = (subset,)

 # needed for mypy since can't narrow types using np.iterable
 subset = cast(Iterable, subset)

 # Verify all columns in subset exist in the queried dataframe
 # Otherwise, raise a KeyError, same as if you try to __getitem__ with a
 # key that doesn't exist.
 diff = Index(subset).difference(self.columns)
 if not diff.empty:
 raise KeyError(diff)

 vals = (col.values for name, col in self.items() if name in subset)
 labels, shape = map(list, zip(*map(f, vals)))

 ids = get_group_index(labels, shape, sort=False, xnull=False)
 return Series(duplicated_int64(ids, keep), index=self.index)

Sorting

@Substitution(**_shared_doc_kwargs)
@Appender(NDFrame.sort_values.__doc__)
def sort_values(
 self,
 by,
 axis=0,
 ascending=True,
 inplace=False,
 kind="quicksort",
 na_position="last",
 ignore_index=False,
):
 inplace = validate_bool_kwarg(inplace, "inplace")
 axis = self._get_axis_number(axis)

 if not isinstance(by, list):
 by = [by]
 if is_sequence(ascending) and len(by) != len(ascending):
 raise ValueError(
 f"Length of ascending ({len(ascending)}) != length of by ({len(by)})"
)
 if len(by) > 1:
 from pandas.core.sorting import lexsort_indexer

 keys = [self._get_label_or_level_values(x, axis=axis) for x in by]
 indexer = lexsort_indexer(keys, orders=ascending, na_position=na_position)
 indexer = ensure_platform_int(indexer)
 else:
 from pandas.core.sorting import nargsort

 by = by[0]
 k = self._get_label_or_level_values(by, axis=axis)

 if isinstance(ascending, (tuple, list)):
 ascending = ascending[0]

```

```

 indexer = nargsort(
 k, kind=kind, ascending=ascending, na_position=na_position
)

 new_data = self._mgr.take(
 indexer, axis=self._get_block_manager_axis(axis), verify=False
)

 if ignore_index:
 new_data.axes[1] = ibase.default_index(len(indexer))

 result = self._constructor(new_data)
 if inplace:
 return self._update_inplace(result)
 else:
 return result.__finalize__(self, method="sort_values")

def sort_index(
 self,
 axis=0,
 level=None,
 ascending: bool = True,
 inplace: bool = False,
 kind: str = "quicksort",
 na_position: str = "last",
 sort_remaining: bool = True,
 ignore_index: bool = False,
):
 """
 Sort object by labels (along an axis).

 Returns a new DataFrame sorted by label if `inplace` argument is
 ``False``, otherwise updates the original DataFrame and returns None.

 Parameters

 axis : {0 or 'index', 1 or 'columns'}, default 0
 The axis along which to sort. The value 0 identifies the rows,
 and 1 identifies the columns.
 level : int or level name or list of ints or list of level names
 If not None, sort on values in specified index level(s).
 ascending : bool or list of bools, default True
 Sort ascending vs. descending. When the index is a MultiIndex the
 sort direction can be controlled for each level individually.
 inplace : bool, default False
 If True, perform operation in-place.
 kind : {'quicksort', 'mergesort', 'heapsort'}, default 'quicksort'
 Choice of sorting algorithm. See also ndarray.np.sort for more
 information. `mergesort` is the only stable algorithm. For
 DataFrames, this option is only applied when sorting on a single
 column or label.
 na_position : {'first', 'last'}, default 'last'
 Puts NaNs at the beginning if `first`; `last` puts NaNs at the end.
 Not implemented for MultiIndex.
 sort_remaining : bool, default True
 If True and sorting by level and index is multilevel, sort by other
 levels too (in order) after sorting by specified level.
 ignore_index : bool, default False
 If True, the resulting axis will be labeled 0, 1, ..., n - 1.

 .. versionadded:: 1.0.0
 """

 Returns

```

```

DataFrame
 The original DataFrame sorted by the labels.

See Also

Series.sort_index : Sort Series by the index.
DataFrame.sort_values : Sort DataFrame by the value.
Series.sort_values : Sort Series by the value.

Examples

>>> df = pd.DataFrame([1, 2, 3, 4, 5], index=[100, 29, 234, 1, 150],
... columns=['A'])
>>> df.sort_index()
 A
1 4
29 2
100 1
150 5
234 3

By default, it sorts in ascending order, to sort in descending order,
use ``ascending=False``

>>> df.sort_index(ascending=False)
 A
234 3
150 5
100 1
29 2
1 4
"""
TODO: this can be combined with Series.sort_index impl as
almost identical

inplace = validate_bool_kwarg(inplace, "inplace")

axis = self._get_axis_number(axis)
labels = self._get_axis(axis)

make sure that the axis is lexsorted to start
if not we need to reconstruct to get the correct indexer
labels = labels._sort_levels_monotonic()
if level is not None:

 new_axis, indexer = labels.sortlevel(
 level, ascending=ascending, sort_remaining=sort_remaining
)

elif isinstance(labels, ABCMultiIndex):
 from pandas.core.sorting import lexsort_indexer

 indexer = lexsort_indexer(
 labels._get_codes_for_sorting(),
 orders=ascending,
 na_position=na_position,
)
else:
 from pandas.core.sorting import nargsort

 # Check monotonic-ness before sort an index
 # GH11080
```

```
 if (ascending and labels.is_monotonic_increasing) or (
 not ascending and labels.is_monotonic_decreasing
):
 if inplace:
 return
 else:
 return self.copy()

 indexer = narsort(
 labels, kind=kind, ascending=ascending, na_position=na_position
)

 baxis = self._get_block_manager_axis(axis)
 new_data = self._mgr.take(indexer, axis=baxis, verify=False)

 # reconstruct axis if needed
 new_data.axes[baxis] = new_data.axes[baxis]._sort_levels_monotonic()

 if ignore_index:
 new_data.axes[1] = ibase.default_index(len(indexer))

 result = self._constructor(new_data)
 if inplace:
 return self._update_inplace(result)
 else:
 return result.__finalize__(self, method="sort_index")

 def value_counts(
 self,
 subset: Optional[Sequence[Label]] = None,
 normalize: bool = False,
 sort: bool = True,
 ascending: bool = False,
):
 """
 Return a Series containing counts of unique rows in the DataFrame.

 .. versionadded:: 1.1.0

 Parameters

 subset : list-like, optional
 Columns to use when counting unique combinations.
 normalize : bool, default False
 Return proportions rather than frequencies.
 sort : bool, default True
 Sort by frequencies.
 ascending : bool, default False
 Sort in ascending order.

 Returns

 Series

 See Also

 Series.value_counts: Equivalent method on Series.

 Notes

 The returned Series will have a MultiIndex with one level per input
 column. By default, rows that contain any NA values are omitted from
 the result. By default, the resulting Series will be in descending
```

```
order so that the first element is the most frequently-occurring row.
```

Examples

-----

```
>>> df = pd.DataFrame({'num_legs': [2, 4, 4, 6],
... 'num_wings': [2, 0, 0, 0]},
... index=['falcon', 'dog', 'cat', 'ant'])
>>> df
 num_legs num_wings
falcon 2 2
dog 4 0
cat 4 0
ant 6 0

>>> df.value_counts()
num_legs num_wings
4 0 2
6 0 1
2 2 1
dtype: int64

>>> df.value_counts(sort=False)
num_legs num_wings
2 2 1
4 0 2
6 0 1
dtype: int64

>>> df.value_counts(ascending=True)
num_legs num_wings
2 2 1
6 0 1
4 0 2
dtype: int64

>>> df.value_counts(normalize=True)
num_legs num_wings
4 0 0.50
6 0 0.25
2 2 0.25
dtype: float64
"""
if subset is None:
 subset = self.columns.tolist()

counts = self.groupby(subset).size()

if sort:
 counts = counts.sort_values(ascending=ascending)
if normalize:
 counts /= counts.sum()

Force MultiIndex for single column
if len(subset) == 1:
 counts.index = MultiIndex.from_arrays(
 [counts.index], names=[counts.index.name]
)

return counts

def nlargest(self, n, columns, keep="first") -> "DataFrame":
 """
 Return the first `n` rows ordered by `columns` in descending order.
```

Return the first `n` rows with the largest values in `columns`, in descending order. The columns that are not specified are returned as well, but not used for ordering.

This method is equivalent to  
``df.sort\_values(columns, ascending=False).head(n)``, but more performant.

#### Parameters

-----

n : int

Number of rows to return.

columns : label or list of labels

Column label(s) to order by.

keep : {'first', 'last', 'all'}, default 'first'

Where there are duplicate values:

- `first` : prioritize the first occurrence(s)
- `last` : prioritize the last occurrence(s)
- ``all`` : do not drop any duplicates, even it means selecting more than `n` items.

.. versionadded:: 0.24.0

#### Returns

-----

DataFrame

The first `n` rows ordered by the given columns in descending order.

#### See Also

-----

DataFrame.nsmallest : Return the first `n` rows ordered by `columns` in ascending order.

DataFrame.sort\_values : Sort DataFrame by the values.

DataFrame.head : Return the first `n` rows without re-ordering.

#### Notes

-----

This function cannot be used with all column types. For example, when specifying columns with `object` or `category` dtypes, ``TypeError`` is raised.

#### Examples

-----

```
>>> df = pd.DataFrame({'population': [59000000, 65000000, 434000,
... 434000, 434000, 337000, 11300,
... 11300, 11300],
... 'GDP': [1937894, 2583560, 12011, 4520, 12128,
... 17036, 182, 38, 311],
... 'alpha-2': ["IT", "FR", "MT", "MV", "BN",
... "IS", "NR", "TV", "AI"]},
... index=["Italy", "France", "Malta",
... "Maldives", "Brunei", "Iceland",
... "Nauru", "Tuvalu", "Anguilla"])
>>> df
```

|          | population | GDP     | alpha-2 |
|----------|------------|---------|---------|
| Italy    | 59000000   | 1937894 | IT      |
| France   | 65000000   | 2583560 | FR      |
| Malta    | 434000     | 12011   | MT      |
| Maldives | 434000     | 4520    | MV      |
| Brunei   | 434000     | 12128   | BN      |

|          |        |       |    |
|----------|--------|-------|----|
| Iceland  | 337000 | 17036 | IS |
| Nauru    | 11300  | 182   | NR |
| Tuvalu   | 11300  | 38    | TV |
| Anguilla | 11300  | 311   | AI |

In the following example, we will use ``nlargest`` to select the three rows having the largest values in column "population".

```
>>> df.nlargest(3, 'population')
 population GDP alpha-2
France 65000000 2583560 FR
Italy 59000000 1937894 IT
Malta 434000 12011 MT
```

When using ``keep='last'``, ties are resolved in reverse order:

```
>>> df.nlargest(3, 'population', keep='last')
 population GDP alpha-2
France 65000000 2583560 FR
Italy 59000000 1937894 IT
Brunei 434000 12128 BN
```

When using ``keep='all'``, all duplicate items are maintained:

```
>>> df.nlargest(3, 'population', keep='all')
 population GDP alpha-2
France 65000000 2583560 FR
Italy 59000000 1937894 IT
Malta 434000 12011 MT
Maldives 434000 4520 MV
Brunei 434000 12128 BN
```

To order by the largest values in column "population" and then "GDP", we can specify multiple columns like in the next example.

```
>>> df.nlargest(3, ['population', 'GDP'])
 population GDP alpha-2
France 65000000 2583560 FR
Italy 59000000 1937894 IT
Brunei 434000 12128 BN
"""
return algorithms.SelectNFrame(self, n=n, keep=keep, columns=columns).nlargest()
```

```
def nsmallest(self, n, columns, keep="first") -> "DataFrame":
```

```
"""
Return the first `n` rows ordered by `columns` in ascending order.
```

Return the first `n` rows with the smallest values in `columns`, in ascending order. The columns that are not specified are returned as well, but not used for ordering.

This method is equivalent to  
``df.sort\_values(columns, ascending=True).head(n)`` , but more performant.

Parameters

-----

n : int

Number of items to retrieve.

columns : list or str

Column name or names to order by.

keep : {'first', 'last', 'all'}, default 'first'

Where there are duplicate values:

```

- ``first`` : take the first occurrence.
- ``last`` : take the last occurrence.
- ``all`` : do not drop any duplicates, even it means
 selecting more than `n` items.

.. versionadded:: 0.24.0

>Returns

DataFrame

See Also

DataFrame.nlargest : Return the first `n` rows ordered by `columns` in
 descending order.
DataFrame.sort_values : Sort DataFrame by the values.
DataFrame.head : Return the first `n` rows without re-ordering.

Examples

>>> df = pd.DataFrame({'population': [59000000, 65000000, 434000,
... 434000, 434000, 337000, 337000,
... 11300, 11300],
... 'GDP': [1937894, 2583560, 12011, 4520, 12128,
... 17036, 182, 38, 311],
... 'alpha-2': ["IT", "FR", "MT", "MV", "BN",
... "IS", "NR", "TV", "AI"]},
... index=["Italy", "France", "Malta",
... "Maldives", "Brunei", "Iceland",
... "Nauru", "Tuvalu", "Anguilla"])
>>> df
 population GDP alpha-2
Italy 59000000 1937894 IT
France 65000000 2583560 FR
Malta 434000 12011 MT
Maldives 434000 4520 MV
Brunei 434000 12128 BN
Iceland 337000 17036 IS
Nauru 337000 182 NR
Tuvalu 11300 38 TV
Anguilla 11300 311 AI

In the following example, we will use ``nsmallest`` to select the
three rows having the smallest values in column "population".

>>> df.nsmallest(3, 'population')
 population GDP alpha-2
Tuvalu 11300 38 TV
Anguilla 11300 311 AI
Iceland 337000 17036 IS

When using ``keep='last'``, ties are resolved in reverse order:

>>> df.nsmallest(3, 'population', keep='last')
 population GDP alpha-2
Anguilla 11300 311 AI
Tuvalu 11300 38 TV
Nauru 337000 182 NR

When using ``keep='all'``, all duplicate items are maintained:

>>> df.nsmallest(3, 'population', keep='all')

```

|          | population | GDP   | alpha-2 |
|----------|------------|-------|---------|
| Tuvalu   | 11300      | 38    | TV      |
| Anguilla | 11300      | 311   | AI      |
| Iceland  | 337000     | 17036 | IS      |
| Nauru    | 337000     | 182   | NR      |

To order by the smallest values in column "population" and then "GDP", we can specify multiple columns like in the next example.

```
>>> df.nsmallest(3, ['population', 'GDP'])
 population GDP alpha-2
Tuvalu 11300 38 TV
Anguilla 11300 311 AI
Nauru 337000 182 NR
"""
return algorithms.SelectNFrame(
 self, n=n, keep=keep, columns=columns
).nsmallest()

def swaplevel(self, i=-2, j=-1, axis=0) -> "DataFrame":
"""
Swap levels i and j in a MultiIndex on a particular axis.

Parameters

i, j : int or str
 Levels of the indices to be swapped. Can pass level name as string.
axis : {0 or 'index', 1 or 'columns'}, default 0
 The axis to swap levels on. 0 or 'index' for row-wise, 1 or
 'columns' for column-wise.

Returns

DataFrame
"""
result = self.copy()

axis = self._get_axis_number(axis)

if not isinstance(result._get_axis(axis), ABCMultiIndex): # pragma: no cover
 raise TypeError("Can only swap levels on a hierarchical axis.")

if axis == 0:
 assert isinstance(result.index, ABCMultiIndex)
 result.index = result.index.swaplevel(i, j)
else:
 assert isinstance(result.columns, ABCMultiIndex)
 result.columns = result.columns.swaplevel(i, j)
return result

def reorder_levels(self, order, axis=0) -> "DataFrame":
"""
Rearrange index levels using input order. May not drop or duplicate levels.

Parameters

order : list of int or list of str
 List representing new level order. Reference level by number
 (position) or by key (label).
axis : {0 or 'index', 1 or 'columns'}, default 0
 Where to reorder levels.

Returns

```

```

DataFrame
"""
axis = self._get_axis_number(axis)
if not isinstance(self._get_axis(axis), ABCMultiIndex): # pragma: no cover
 raise TypeError("Can only reorder levels on a hierarchical axis.")

result = self.copy()

if axis == 0:
 assert isinstance(result.index, ABCMultiIndex)
 result.index = result.index.reorder_levels(order)
else:
 assert isinstance(result.columns, ABCMultiIndex)
 result.columns = result.columns.reorder_levels(order)
return result

Arithmetic / combination related

def _combine_frame(self, other: "DataFrame", func, fill_value=None):
 # at this point we have `self._indexed_same(other)`

 if fill_value is None:
 # since _arith_op may be called in a loop, avoid function call
 # overhead if possible by doing this check once
 _arith_op = func

 else:

 def _arith_op(left, right):
 # for the mixed_type case where we iterate over columns,
 # _arith_op(left, right) is equivalent to
 # left._binop(right, func, fill_value=fill_value)
 left, right = ops.fill_binop(left, right, fill_value)
 return func(left, right)

 if ops.should_series_dispatch(self, other, func):
 # iterate over columns
 new_data = ops.dispatch_to_series(self, other, _arith_op)
 else:
 with np.errstate(all="ignore"):
 res_values = _arith_op(self.values, other.values)
 new_data = dispatch_fill_zeros(func, self.values, other.values, res_values)

 return new_data

def _construct_result(self, result) -> "DataFrame":
 """
 Wrap the result of an arithmetic, comparison, or logical operation.

 Parameters

 result : DataFrame

 Returns

 DataFrame
 """
 out = self._constructor(result, index=self.index, copy=False)
 # Pin columns instead of passing to constructor for compat with
 # non-unique columns case
 out.columns = self.columns

```

```
 return out

def combine(
 self, other: "DataFrame", func, fill_value=None, overwrite=True
) -> "DataFrame":
 """
 Perform column-wise combine with another DataFrame.

 Combines a DataFrame with `other` DataFrame using `func` to element-wise
 combine columns. The row and column indexes of the resulting DataFrame
 will be the union of the two.

 Parameters

 other : DataFrame
 The DataFrame to merge column-wise.
 func : function
 Function that takes two series as inputs and return a Series or a
 scalar. Used to merge the two dataframes column by columns.
 fill_value : scalar value, default None
 The value to fill NaNs with prior to passing any column to the
 merge func.
 overwrite : bool, default True
 If True, columns in `self` that do not exist in `other` will be
 overwritten with NaNs.

 Returns

 DataFrame
 Combination of the provided DataFrames.

 See Also

 DataFrame.combine_first : Combine two DataFrame objects and default to
 non-null values in frame calling the method.

 Examples

 Combine using a simple function that chooses the smaller column.

 >>> df1 = pd.DataFrame({'A': [0, 0], 'B': [4, 4]})
 >>> df2 = pd.DataFrame({'A': [1, 1], 'B': [3, 3]})
 >>> take_smaller = lambda s1, s2: s1 if s1.sum() < s2.sum() else s2
 >>> df1.combine(df2, take_smaller)
 A B
 0 0 3
 1 0 3

 Example using a true element-wise combine function.

 >>> df1 = pd.DataFrame({'A': [5, 0], 'B': [2, 4]})
 >>> df2 = pd.DataFrame({'A': [1, 1], 'B': [3, 3]})
 >>> df1.combine(df2, np.minimum)
 A B
 0 1 2
 1 0 3

 Using `fill_value` fills Nones prior to passing the column to the
 merge function.

 >>> df1 = pd.DataFrame({'A': [0, 0], 'B': [None, 4]})
 >>> df2 = pd.DataFrame({'A': [1, 1], 'B': [3, 3]})
 >>> df1.combine(df2, take_smaller, fill_value=-5)
```

```
A B
0 0 -5.0
1 0 4.0
```

However, if the same element in both dataframes is None, that None is preserved

```
>>> df1 = pd.DataFrame({'A': [0, 0], 'B': [None, 4]})
>>> df2 = pd.DataFrame({'A': [1, 1], 'B': [None, 3]})
>>> df1.combine(df2, take_smaller, fill_value=-5)
A B
0 0 -5.0
1 0 3.0
```

Example that demonstrates the use of `overwrite` and behavior when the axis differ between the dataframes.

```
>>> df1 = pd.DataFrame({'A': [0, 0], 'B': [4, 4]})
>>> df2 = pd.DataFrame({'B': [3, 3], 'C': [-10, 1], }, index=[1, 2])
>>> df1.combine(df2, take_smaller)
A B C
0 NaN NaN NaN
1 NaN 3.0 -10.0
2 NaN 3.0 1.0

>>> df1.combine(df2, take_smaller, overwrite=False)
A B C
0 0.0 NaN NaN
1 0.0 3.0 -10.0
2 NaN 3.0 1.0
```

Demonstrating the preference of the passed in dataframe.

```
>>> df2 = pd.DataFrame({'B': [3, 3], 'C': [1, 1], }, index=[1, 2])
>>> df2.combine(df1, take_smaller)
A B C
0 0.0 NaN NaN
1 0.0 3.0 NaN
2 NaN 3.0 NaN

>>> df2.combine(df1, take_smaller, overwrite=False)
A B C
0 0.0 NaN NaN
1 0.0 3.0 1.0
2 NaN 3.0 1.0
"""
other_idxlen = len(other.index) # save for compare

this, other = self.align(other, copy=False)
new_index = this.index

if other.empty and len(new_index) == len(self.index):
 return self.copy()

if self.empty and len(other) == other_idxlen:
 return other.copy()

sorts if possible
new_columns = this.columns.union(other.columns)
do_fill = fill_value is not None
result = {}
for col in new_columns:
 series = this[col]
```

```

otherSeries = other[col]

this_dtype = series.dtype
other_dtype = otherSeries.dtype

this_mask = isna(series)
other_mask = isna(otherSeries)

don't overwrite columns unnecessarily
DO propagate if this column is not in the intersection
if not overwrite and other_mask.all():
 result[col] = this[col].copy()
 continue

if do_fill:
 series = series.copy()
 otherSeries = otherSeries.copy()
 series[this_mask] = fill_value
 otherSeries[other_mask] = fill_value

if col not in self.columns:
 # If self DataFrame does not have col in other DataFrame,
 # try to promote series, which is all NaN, as other_dtype.
 new_dtype = other_dtype
 try:
 series = series.astype(new_dtype, copy=False)
 except ValueError:
 # e.g. new_dtype is integer types
 pass
else:
 # if we have different dtypes, possibly promote
 new_dtype = find_common_type([this_dtype, other_dtype])
 if not is_dtype_equal(this_dtype, new_dtype):
 series = series.astype(new_dtype)
 if not is_dtype_equal(other_dtype, new_dtype):
 otherSeries = otherSeries.astype(new_dtype)

arr = func(series, otherSeries)
arr = maybe_downcast_to_dtype(arr, this_dtype)

result[col] = arr

convert_objects just in case
return self._constructor(result, index=new_index, columns=new_columns)

def combine_first(self, other: "DataFrame") -> "DataFrame":
 """
 Update null elements with value in the same location in `other`.

 Combine two DataFrame objects by filling null values in one DataFrame
 with non-null values from other DataFrame. The row and column indexes
 of the resulting DataFrame will be the union of the two.

 Parameters

 other : DataFrame
 Provided DataFrame to use to fill null values.

 Returns

 DataFrame

 See Also

```

```

DataFrame.combine : Perform series-wise operation on two DataFrames
 using a given function.

Examples

>>> df1 = pd.DataFrame({'A': [None, 0], 'B': [None, 4]})
>>> df2 = pd.DataFrame({'A': [1, 1], 'B': [3, 3]})
>>> df1.combine_first(df2)
 A B
0 1.0 3.0
1 0.0 4.0

Null values still persist if the location of that null value
does not exist in `other`'s index

>>> df1 = pd.DataFrame({'A': [None, 0], 'B': [4, None]})
>>> df2 = pd.DataFrame({'B': [3, 3], 'C': [1, 1]}, index=[1, 2])
>>> df1.combine_first(df2)
 A B C
0 NaN 4.0 NaN
1 0.0 3.0 1.0
2 NaN 3.0 1.0
"""
import pandas.core.computation.expressions as expressions

def extract_values(arr):
 # Does two things:
 # 1. maybe gets the values from the Series / Index
 # 2. convert datelike to i8
 if isinstance(arr, (ABCIndexClass, ABCSeries)):
 arr = arr._values

 if needs_i8_conversion(arr):
 if is_extension_array_dtype(arr.dtype):
 arr = arr.astype('i8')
 else:
 arr = arr.view('i8')
 return arr

def combiner(x, y):
 mask = isna(x)
 if isinstance(mask, (ABCIndexClass, ABCSeries)):
 mask = mask._values

 x_values = extract_values(x)
 y_values = extract_values(y)

 # If the column y in other DataFrame is not in first DataFrame,
 # just return y_values.
 if y.name not in self.columns:
 return y_values

 return expressions.where(mask, y_values, x_values)

return self.combine(other, combiner, overwrite=False)

def update(
 self, other, join="left", overwrite=True, filter_func=None, errors="ignore"
) -> None:
 """
 Modify in place using non-NA values from another DataFrame.

```

Aligns on indices. There is no return value.

#### Parameters

-----

other : DataFrame, or object coercible into a DataFrame  
Should have at least one matching index/column label  
with the original DataFrame. If a Series is passed,  
its name attribute must be set, and that will be  
used as the column name to align with the original DataFrame.

join : {'left'}, default 'left'  
Only left join is implemented, keeping the index and columns of the  
original object.

overwrite : bool, default True  
How to handle non-NA values for overlapping keys:

- \* True: overwrite original DataFrame's values  
with values from `other`.
- \* False: only update values that are NA in  
the original DataFrame.

filter\_func : callable(1d-array) -> bool 1d-array, optional  
Can choose to replace values other than NA. Return True for values  
that should be updated.

errors : {'raise', 'ignore'}, default 'ignore'  
If 'raise', will raise a ValueError if the DataFrame and `other`  
both contain non-NA data in the same place.

.. versionchanged:: 0.24.0  
Changed from `raise\_conflict=False|True`  
to `errors='ignore'|'raise'`.

#### Returns

-----

None : method directly changes calling object

#### Raises

-----

##### ValueError

- \* When `errors='raise'` and there's overlapping non-NA data.
- \* When `errors` is not either ``'ignore'`` or ``'raise'``

##### NotImplementedError

- \* If `join != 'left'`

#### See Also

-----

dict.update : Similar method for dictionaries.

DataFrame.merge : For column(s)-on-column(s) operations.

#### Examples

-----

```
>>> df = pd.DataFrame({'A': [1, 2, 3],
... 'B': [400, 500, 600]})
>>> new_df = pd.DataFrame({'B': [4, 5, 6],
... 'C': [7, 8, 9]})
>>> df.update(new_df)
>>> df
 A B
0 1 4
1 2 5
2 3 6
```

The DataFrame's length does not increase as a result of the update,  
only values at matching index/column labels are updated.

```
>>> df = pd.DataFrame({'A': ['a', 'b', 'c'],
... 'B': ['x', 'y', 'z']})
>>> new_df = pd.DataFrame({'B': ['d', 'e', 'f', 'g', 'h', 'i']})
>>> df.update(new_df)
>>> df
 A B
0 a d
1 b e
2 c f
```

For Series, it's name attribute must be set.

```
>>> df = pd.DataFrame({'A': ['a', 'b', 'c'],
... 'B': ['x', 'y', 'z']})
>>> new_column = pd.Series(['d', 'e'], name='B', index=[0, 2])
>>> df.update(new_column)
>>> df
 A B
0 a d
1 b y
2 c e
>>> df = pd.DataFrame({'A': ['a', 'b', 'c'],
... 'B': ['x', 'y', 'z']})
>>> new_df = pd.DataFrame({'B': ['d', 'e']}, index=[1, 2])
>>> df.update(new_df)
>>> df
 A B
0 a x
1 b d
2 c e
```

If `other` contains NaNs the corresponding values are not updated in the original dataframe.

```
>>> df = pd.DataFrame({'A': [1, 2, 3],
... 'B': [400, 500, 600]})
>>> new_df = pd.DataFrame({'B': [4, np.nan, 6]})
>>> df.update(new_df)
>>> df
 A B
0 1 4.0
1 2 500.0
2 3 6.0
"""
import pandas.core.computation.expressions as expressions

TODO: Support other joins
if join != "left": # pragma: no cover
 raise NotImplementedError("Only left join is supported")
if errors not in ["ignore", "raise"]:
 raise ValueError("The parameter errors must be either 'ignore' or 'raise'")

if not isinstance(other, DataFrame):
 other = DataFrame(other)

other = other.reindex_like(self)

for col in self.columns:
 this = self[col]._values
 that = other[col]._values
 if filter_func is not None:
 with np.errstate(all="ignore"):
```

```

 mask = ~filter_func(this) | isna(that)
 else:
 if errors == "raise":
 mask_this = notna(that)
 mask_that = notna(this)
 if any(mask_this & mask_that):
 raise ValueError("Data overlaps.")

 if overwrite:
 mask = isna(that)
 else:
 mask = notna(this)

 # don't overwrite columns unnecessarily
 if mask.all():
 continue

 self[col] = expressions.where(mask, this, that)

Data reshaping
@Appender(
"""
Examples

>>> df = pd.DataFrame({'Animal': ['Falcon', 'Falcon',
... 'Parrot', 'Parrot'],
... 'Max Speed': [380., 370., 24., 26.]})
>>> df
 Animal Max Speed
0 Falcon 380.0
1 Falcon 370.0
2 Parrot 24.0
3 Parrot 26.0
>>> df.groupby(['Animal']).mean()
 Max Speed
Animal
Falcon 375.0
Parrot 25.0

Hierarchical Indexes

We can groupby different levels of a hierarchical index
using the `level` parameter:

>>> arrays = [['Falcon', 'Falcon', 'Parrot', 'Parrot'],
... ['Captive', 'Wild', 'Captive', 'Wild']]
>>> index = pd.MultiIndex.from_arrays(arrays, names=('Animal', 'Type'))
>>> df = pd.DataFrame({'Max Speed': [390., 350., 30., 20.]},
... index=index)
>>> df
 Max Speed
Animal Type
Falcon Captive 390.0
 Wild 350.0
Parrot Captive 30.0
 Wild 20.0
>>> df.groupby(level=0).mean()
 Max Speed
Animal
Falcon 370.0
Parrot 25.0
>>> df.groupby(level="Type").mean()

```

```

 Max Speed
Type
Captive 210.0
Wild 185.0
"""
)
@Appender(_shared_docs["groupby"] % _shared_doc_kwargs)
def groupby(
 self,
 by=None,
 axis=0,
 level=None,
 as_index: bool = True,
 sort: bool = True,
 group_keys: bool = True,
 squeeze: bool = False,
 observed: bool = False,
) -> "DataFrameGroupBy":
 from pandas.core.groupby.generic import DataFrameGroupBy

 if level is None and by is None:
 raise TypeError("You have to supply one of 'by' and 'level'")
 axis = self._get_axis_number(axis)

 return DataFrameGroupBy(
 obj=self,
 keys=by,
 axis=axis,
 level=level,
 as_index=as_index,
 sort=sort,
 group_keys=group_keys,
 squeeze=squeeze,
 observed=observed,
)

_shared_docs[
 "pivot"
] = """
Return reshaped DataFrame organized by given index / column values.

Reshape data (produce a "pivot" table) based on column values. Uses unique values from specified `index` / `columns` to form axes of the resulting DataFrame. This function does not support data aggregation, multiple values will result in a MultiIndex in the columns. See the :ref:`User Guide <reshaping>` for more on reshaping.

Parameters

index : str or object or a list of str, optional
 Column to use to make new frame's index. If None, uses existing index.

 .. versionchanged:: 1.1.0
 Also accept list of index names.

columns : str or object or a list of str
 Column to use to make new frame's columns.

 .. versionchanged:: 1.1.0
 Also accept list of columns names.

values : str, object or a list of the previous, optional
"""

```

Column(s) to use for populating new frame's values. If not specified, all remaining columns will be used and the result will have hierarchically indexed columns.

.. versionchanged:: 0.23.0  
Also accept list of column names.

Returns

-----

DataFrame

Returns reshaped DataFrame.

Raises

-----

ValueError:

When there are any `index`, `columns` combinations with multiple values. `DataFrame.pivot\_table` when you need to aggregate.

See Also

-----

DataFrame.pivot\_table : Generalization of pivot that can handle duplicate values for one index/column pair.

DataFrame.unstack : Pivot based on the index values instead of a column.

Notes

-----

For finer-tuned control, see hierarchical indexing documentation along with the related stack/unstack methods.

Examples

-----

```
>>> df = pd.DataFrame({'foo': ['one', 'one', 'one', 'two', 'two',
... 'two'],
... 'bar': ['A', 'B', 'C', 'A', 'B', 'C'],
... 'baz': [1, 2, 3, 4, 5, 6],
... 'zoo': ['x', 'y', 'z', 'q', 'w', 't']})
```

>>> df

|   | foo | bar | baz | zoo |
|---|-----|-----|-----|-----|
| 0 | one | A   | 1   | x   |
| 1 | one | B   | 2   | y   |
| 2 | one | C   | 3   | z   |
| 3 | two | A   | 4   | q   |
| 4 | two | B   | 5   | w   |
| 5 | two | C   | 6   | t   |

```
>>> df.pivot(index='foo', columns='bar', values='baz')
```

```
bar A B C
```

```
foo
```

|     |   |   |   |
|-----|---|---|---|
| one | 1 | 2 | 3 |
| two | 4 | 5 | 6 |

```
>>> df.pivot(index='foo', columns='bar')['baz']
```

```
bar A B C
```

```
foo
```

|     |   |   |   |
|-----|---|---|---|
| one | 1 | 2 | 3 |
| two | 4 | 5 | 6 |

```
>>> df.pivot(index='foo', columns='bar', values=['baz', 'zoo'])
```

```
 baz zoo
```

```
bar A B C A B C
```

```
foo
```

|     |   |   |   |   |   |   |
|-----|---|---|---|---|---|---|
| one | 1 | 2 | 3 | x | y | z |
|-----|---|---|---|---|---|---|

```
two 4 5 6 q w t
```

You could also assign a list of column names or a list of index names.

```
>>> df = pd.DataFrame({
... "lev1": [1, 1, 1, 2, 2, 2],
... "lev2": [1, 1, 2, 1, 1, 2],
... "lev3": [1, 2, 1, 2, 1, 2],
... "lev4": [1, 2, 3, 4, 5, 6],
... "values": [0, 1, 2, 3, 4, 5]})

>>> df
 lev1 lev2 lev3 lev4 values
0 1 1 1 1 0
1 1 1 2 2 1
2 1 2 1 3 2
3 2 1 2 4 3
4 2 1 1 5 4
5 2 2 2 6 5

>>> df.pivot(index="lev1", columns=["lev2", "lev3"], values="values")
lev2 1 2
lev3 1 2
lev1
1 0.0 1.0 2.0 NaN
2 4.0 3.0 NaN 5.0

>>> df.pivot(index=["lev1", "lev2"], columns=["lev3"], values="values")
 lev3 1 2
lev1 lev2
1 1 0.0 1.0
2 2 2.0 NaN
2 1 4.0 3.0
2 2 NaN 5.0
```

A ValueError is raised if there are any duplicates.

```
>>> df = pd.DataFrame({"foo": ['one', 'one', 'two', 'two'],
... "bar": ['A', 'A', 'B', 'C'],
... "baz": [1, 2, 3, 4]})

>>> df
 foo bar baz
0 one A 1
1 one A 2
2 two B 3
3 two C 4
```

Notice that the first two rows are the same for our `index` and `columns` arguments.

```
>>> df.pivot(index='foo', columns='bar', values='baz')
Traceback (most recent call last):
...
ValueError: Index contains duplicate entries, cannot reshape
"""
```

```
@Substitution("")
@Appender(_shared_docs["pivot"])
def pivot(self, index=None, columns=None, values=None) -> "DataFrame":
 from pandas.core.reshape.pivot import pivot

 return pivot(self, index=index, columns=columns, values=values)

_shared_docs[
```

```
"pivot_table"
] = """
Create a spreadsheet-style pivot table as a DataFrame.

The levels in the pivot table will be stored in MultiIndex objects
(hierarchical indexes) on the index and columns of the result DataFrame.

Parameters

values : column to aggregate, optional
index : column, Grouper, array, or list of the previous
 If an array is passed, it must be the same length as the data. The
 list can contain any of the other types (except list).
 Keys to group by on the pivot table index. If an array is passed,
 it is being used as the same manner as column values.
columns : column, Grouper, array, or list of the previous
 If an array is passed, it must be the same length as the data. The
 list can contain any of the other types (except list).
 Keys to group by on the pivot table column. If an array is passed,
 it is being used as the same manner as column values.
aggfunc : function, list of functions, dict, default numpy.mean
 If list of functions passed, the resulting pivot table will have
 hierarchical columns whose top level are the function names
 (inferred from the function objects themselves)
 If dict is passed, the key is column to aggregate and value
 is function or list of functions.
fill_value : scalar, default None
 Value to replace missing values with (in the resulting pivot table,
 after aggregation).
margins : bool, default False
 Add all row / columns (e.g. for subtotal / grand totals).
dropna : bool, default True
 Do not include columns whose entries are all NaN.
margins_name : str, default 'All'
 Name of the row / column that will contain the totals
 when margins is True.
observed : bool, default False
 This only applies if any of the groupers are Categoricals.
 If True: only show observed values for categorical groupers.
 If False: show all values for categorical groupers.

.. versionchanged:: 0.25.0

Returns

DataFrame
 An Excel style pivot table.

See Also

DataFrame.pivot : Pivot without aggregation that can handle
 non-numeric data.

Examples

>>> df = pd.DataFrame({'A': ['foo', 'foo', 'foo', 'foo', 'foo',
... 'bar', 'bar', 'bar', 'bar'],
... 'B': ['one', 'one', 'one', 'two', 'two',
... 'one', 'one', 'two', 'two'],
... 'C': ['small', 'large', 'large', 'small',
... 'small', 'large', 'small', 'small',
... 'large'],
... 'D': [1, 2, 2, 3, 3, 4, 5, 6, 7],
```

```

...
 "E": [2, 4, 5, 5, 6, 6, 8, 9, 9]})

>>> df
 A B C D E
0 foo one small 1 2
1 foo one large 2 4
2 foo one large 2 5
3 foo two small 3 5
4 foo two small 3 6
5 bar one large 4 6
6 bar one small 5 8
7 bar two small 6 9
8 bar two large 7 9

```

This first example aggregates values by taking the sum.

```

>>> table = pd.pivot_table(df, values='D', index=['A', 'B'],
... columns=['C'], aggfunc=np.sum)
>>> table
C large small
A B
bar one 4.0 5.0
 two 7.0 6.0
foo one 4.0 1.0
 two NaN 6.0

```

We can also fill missing values using the `fill\_value` parameter.

```

>>> table = pd.pivot_table(df, values='D', index=['A', 'B'],
... columns=['C'], aggfunc=np.sum, fill_value=0)
>>> table
C large small
A B
bar one 4 5
 two 7 6
foo one 4 1
 two 0 6

```

The next example aggregates by taking the mean across multiple columns.

```

>>> table = pd.pivot_table(df, values=['D', 'E'], index=['A', 'C'],
... aggfunc={'D': np.mean,
... 'E': np.mean})
>>> table
 D E
A C
bar large 5.500000 7.500000
 small 5.500000 8.500000
foo large 2.000000 4.500000
 small 2.333333 4.333333

```

We can also calculate multiple types of aggregations for any given value column.

```

>>> table = pd.pivot_table(df, values=['D', 'E'], index=['A', 'C'],
... aggfunc={'D': np.mean,
... 'E': [min, max, np.mean]})
>>> table
 D E
 mean max mean min
A C
bar large 5.500000 9.0 7.500000 6.0
 small 5.500000 9.0 8.500000 8.0
foo large 2.000000 5.0 4.500000 4.0

```

```

 small 2.333333 6.0 4.333333 2.0
"""

@Substitution("")
@Appender(_shared_docs["pivot_table"])
def pivot_table(
 self,
 values=None,
 index=None,
 columns=None,
 aggfunc="mean",
 fill_value=None,
 margins=False,
 dropna=True,
 margins_name="All",
 observed=False,
) -> "DataFrame":
 from pandas.core.reshape.pivot import pivot_table

 return pivot_table(
 self,
 values=values,
 index=index,
 columns=columns,
 aggfunc=aggfunc,
 fill_value=fill_value,
 margins=margins,
 dropna=dropna,
 margins_name=margins_name,
 observed=observed,
)

def stack(self, level=-1, dropna=True):
 """
 Stack the prescribed level(s) from columns to index.

 Return a reshaped DataFrame or Series having a multi-level
 index with one or more new inner-most levels compared to the current
 DataFrame. The new inner-most levels are created by pivoting the
 columns of the current dataframe:

 - if the columns have a single level, the output is a Series;
 - if the columns have multiple levels, the new index
 level(s) is (are) taken from the prescribed level(s) and
 the output is a DataFrame.

 The new index levels are sorted.

 Parameters

 level : int, str, list, default -1
 Level(s) to stack from the column axis onto the index
 axis, defined as one index or label, or a list of indices
 or labels.
 dropna : bool, default True
 Whether to drop rows in the resulting Frame/Series with
 missing values. Stacking a column level onto the index
 axis can create combinations of index and column values
 that are missing from the original dataframe. See Examples
 section.

 Returns

 """

```

```
DataFrame or Series
 Stacked dataframe or series.

See Also

DataFrame.unstack : Unstack prescribed level(s) from index axis
 onto column axis.
DataFrame.pivot : Reshape dataframe from long format to wide
 format.
DataFrame.pivot_table : Create a spreadsheet-style pivot table
 as a DataFrame.
```

#### Notes

-----  
The function is named by analogy with a collection of books being reorganized from being side by side on a horizontal position (the columns of the dataframe) to being stacked vertically on top of each other (in the index of the dataframe).

#### Examples

```

Single level columns

>>> df_single_level_cols = pd.DataFrame([[0, 1], [2, 3]],
... index=['cat', 'dog'],
... columns=['weight', 'height'])
```

Stacking a dataframe with a single level column axis returns a Series:

```
>>> df_single_level_cols
 weight height
cat 0 1
dog 2 3
>>> df_single_level_cols.stack()
cat weight 0
 height 1
dog weight 2
 height 3
dtype: int64
```

\*\*Multi level columns: simple case\*\*

```
>>> multicol1 = pd.MultiIndex.from_tuples([('weight', 'kg'),
... ('weight', 'pounds')])
>>> df_multi_level_cols1 = pd.DataFrame([[1, 2], [2, 4]],
... index=['cat', 'dog'],
... columns=multicol1)
```

Stacking a dataframe with a multi-level column axis:

```
>>> df_multi_level_cols1
 weight
 kg pounds
cat 1 2
dog 2 4
>>> df_multi_level_cols1.stack()
 weight
cat kg 1
 pounds 2
dog kg 2
 pounds 4
```

```
Missing values

>>> multicol2 = pd.MultiIndex.from_tuples([('weight', 'kg'),
... ('height', 'm')])
>>> df_multi_level_cols2 = pd.DataFrame([[1.0, 2.0], [3.0, 4.0]],
... index=['cat', 'dog'],
... columns=multicol2)
```

It is common to have missing values when stacking a dataframe with multi-level columns, as the stacked dataframe typically has more values than the original dataframe. Missing values are filled with NaNs:

```
>>> df_multi_level_cols2
 weight height
 kg m
cat 1.0 2.0
dog 3.0 4.0
>>> df_multi_level_cols2.stack()
 height weight
cat kg NaN 1.0
 m 2.0 NaN
dog kg NaN 3.0
 m 4.0 NaN
```

\*\*Prescribing the level(s) to be stacked\*\*

The first parameter controls which level or levels are stacked:

```
>>> df_multi_level_cols2.stack(0)
 kg m
cat height NaN 2.0
 weight 1.0 NaN
dog height NaN 4.0
 weight 3.0 NaN
>>> df_multi_level_cols2.stack([0, 1])
cat height m 2.0
 weight kg 1.0
dog height m 4.0
 weight kg 3.0
dtype: float64
```

\*\*Dropping missing values\*\*

```
>>> df_multi_level_cols3 = pd.DataFrame([[None, 1.0], [2.0, 3.0]],
... index=['cat', 'dog'],
... columns=multicol2)
```

Note that rows where all values are missing are dropped by default but this behaviour can be controlled via the dropna keyword parameter:

```
>>> df_multi_level_cols3
 weight height
 kg m
cat NaN 1.0
dog 2.0 3.0
>>> df_multi_level_cols3.stack(dropna=False)
 height weight
cat kg NaN NaN
 m 1.0 NaN
dog kg NaN 2.0
 m 3.0 NaN
```

```
>>> df_multi_level_cols3.stack(dropna=True)
 height weight
cat m 1.0 NaN
dog kg NaN 2.0
 m 3.0 NaN
"""
from pandas.core.reshape.reshape import stack, stack_multiple

if isinstance(level, (tuple, list)):
 return stack_multiple(self, level, dropna=dropna)
else:
 return stack(self, level, dropna=dropna)

def explode(self, column: Union[str, Tuple]) -> "DataFrame":
 """
 Transform each element of a list-like to a row, replicating index values.

 .. versionadded:: 0.25.0

 Parameters

 column : str or tuple
 Column to explode.

 Returns

 DataFrame
 Exploded lists to rows of the subset columns;
 index will be duplicated for these rows.

 Raises

 ValueError :
 if columns of the frame are not unique.

 See Also

 DataFrame.unstack : Pivot a level of the (necessarily hierarchical)
 index labels.
 DataFrame.melt : Unpivot a DataFrame from wide format to long format.
 Series.explode : Explode a DataFrame from list-like columns to long format.

 Notes

 This routine will explode list-likes including lists, tuples,
 Series, and np.ndarray. The result dtype of the subset rows will
 be object. Scalars will be returned unchanged. Empty list-likes will
 result in a np.nan for that row.

 Examples

 >>> df = pd.DataFrame({'A': [[1, 2, 3], 'foo', [], [3, 4]], 'B': 1})
 >>> df
 A B
 0 [1, 2, 3] 1
 1 foo 1
 2 [] 1
 3 [3, 4] 1

 >>> df.explode('A')
 A B
 0 1 1
 0 2 1
```

```

0 3 1
1 foo 1
2 NaN 1
3 3 1
3 4 1
"""
if not (is_scalar(column) or isinstance(column, tuple)):
 raise ValueError("column must be a scalar")
if not self.columns.is_unique:
 raise ValueError("columns must be unique")

df = self.reset_index(drop=True)
TODO: use overload to refine return type of reset_index
assert df is not None # needed for mypy
result = df[column].explode()
result = df.drop([column], axis=1).join(result)
result.index = self.index.take(result.index)
result = result.reindex(columns=self.columns, copy=False)

return result

def unstack(self, level=-1, fill_value=None):
 """
 Pivot a level of the (necessarily hierarchical) index labels.

 Returns a DataFrame having a new level of column labels whose inner-most level
 consists of the pivoted index labels.

 If the index is not a MultiIndex, the output will be a Series
 (the analogue of stack when the columns are not a MultiIndex).

 The level involved will automatically get sorted.

 Parameters

 level : int, str, or list of these, default -1 (last level)
 Level(s) of index to unstack, can pass level name.
 fill_value : int, str or dict
 Replace NaN with this value if the unstack produces missing values.

 Returns

 Series or DataFrame

 See Also

 DataFrame.pivot : Pivot a table based on column values.
 DataFrame.stack : Pivot a level of the column labels (inverse operation
 from `unstack`).

 Examples

 >>> index = pd.MultiIndex.from_tuples([('one', 'a'), ('one', 'b'),
... ('two', 'a'), ('two', 'b')])
 >>> s = pd.Series(np.arange(1.0, 5.0), index=index)
 >>> s
 one a 1.0
 b 2.0
 two a 3.0
 b 4.0
 dtype: float64

 >>> s.unstack(level=-1)

```

```
 a b
one 1.0 2.0
two 3.0 4.0

>>> s.unstack(level=0)
 one two
a 1.0 3.0
b 2.0 4.0

>>> df = s.unstack(level=0)
>>> df.unstack()
one a 1.0
 b 2.0
two a 3.0
 b 4.0
dtype: float64
"""
from pandas.core.reshape.reshape import unstack

return unstack(self, level, fill_value)
```

\_shared\_docs[  
 "melt"] = """

Unpivot a DataFrame from wide to long format, optionally leaving identifiers set.

This function is useful to massage a DataFrame into a format where one or more columns are identifier variables ('id\_vars'), while all other columns, considered measured variables ('value\_vars'), are "unpivoted" to the row axis, leaving just two non-identifier columns, 'variable' and 'value'.

%(versionadded)s

Parameters

-----

id\_vars : tuple, list, or ndarray, optional  
Column(s) to use as identifier variables.

value\_vars : tuple, list, or ndarray, optional  
Column(s) to unpivot. If not specified, uses all columns that are not set as 'id\_vars'.

var\_name : scalar  
Name to use for the 'variable' column. If None it uses ``frame.columns.name`` or 'variable'.

value\_name : scalar, default 'value'  
Name to use for the 'value' column.

col\_level : int or str, optional  
If columns are a MultiIndex then use this level to melt.

Returns

-----

DataFrame  
    Unpivoted DataFrame.

See Also

-----

%(other)s : Identical method.

pivot\_table : Create a spreadsheet-style pivot table as a DataFrame.

DataFrame.pivot : Return reshaped DataFrame organized  
    by given index / column values.

DataFrame.explode : Explode a DataFrame from list-like  
    columns to long format.

Examples

-----

```

>>> df = pd.DataFrame({'A': {0: 'a', 1: 'b', 2: 'c'},
... 'B': {0: 1, 1: 3, 2: 5},
... 'C': {0: 2, 1: 4, 2: 6}})
>>> df
 A B C
0 a 1 2
1 b 3 4
2 c 5 6

>>> %(caller)sid_vars=['A'], value_vars=['B'])
 A variable value
0 a B 1
1 b B 3
2 c B 5

>>> %(caller)sid_vars=['A'], value_vars=['B', 'C'])
 A variable value
0 a B 1
1 b B 3
2 c B 5
3 a C 2
4 b C 4
5 c C 6

```

The names of 'variable' and 'value' columns can be customized:

```

>>> %(caller)sid_vars=['A'], value_vars=['B'],
... var_name='myVarname', value_name='myValname')
 A myVarname myValname
0 a B 1
1 b B 3
2 c B 5

```

If you have multi-index columns:

```

>>> df.columns = [list('ABC'), list('DEF')]
>>> df
 A B C
 D E F
0 a 1 2
1 b 3 4
2 c 5 6

>>> %(caller)scol_level=0, id_vars=['A'], value_vars=['B'])
 A variable value
0 a B 1
1 b B 3
2 c B 5

>>> %(caller)sid_vars=[('A', 'D')], value_vars=[('B', 'E')])
 (A, D) variable_0 variable_1 value
0 a B E 1
1 b B E 3
2 c B E 5
"""


```

```

@Appender(
 _shared_docs["melt"]
 % dict(
 caller="df.melt(\"",
 versionadded="\n .. versionadded:: 0.20.0\n",
 other="melt",
)
)

```

```


)

def melt(
 self,
 id_vars=None,
 value_vars=None,
 var_name=None,
 value_name="value",
 col_level=None,
) -> "DataFrame":
 from pandas.core.reshape.melt import melt

 return melt(
 self,
 id_vars=id_vars,
 value_vars=value_vars,
 var_name=var_name,
 value_name=value_name,
 col_level=col_level,
)

Time series-related

def diff(self, periods: int = 1, axis: Axis = 0) -> "DataFrame":
 """
 First discrete difference of element.

 Calculates the difference of a DataFrame element compared with another
 element in the DataFrame (default is the element in the same column
 of the previous row).

 Parameters

 periods : int, default 1
 Periods to shift for calculating difference, accepts negative
 values.
 axis : {0 or 'index', 1 or 'columns'}, default 0
 Take difference over rows (0) or columns (1).

 Returns

 DataFrame

 See Also

 Series.diff: First discrete difference for a Series.
 DataFrame.pct_change: Percent change over given number of periods.
 DataFrame.shift: Shift index by desired number of periods with an
 optional time freq.

 Notes

 For boolean dtypes, this uses :meth:`operator.xor` rather than
 :meth:`operator.sub`.

 Examples

 Difference with previous row

>>> df = pd.DataFrame({'a': [1, 2, 3, 4, 5, 6],
... 'b': [1, 1, 2, 3, 5, 8],
... 'c': [1, 4, 9, 16, 25, 36]})
>>> df


```

```

 a b c
0 1 1 1
1 2 1 4
2 3 2 9
3 4 3 16
4 5 5 25
5 6 8 36

>>> df.diff()
 a b c
0 NaN NaN NaN
1 1.0 0.0 3.0
2 1.0 1.0 5.0
3 1.0 1.0 7.0
4 1.0 2.0 9.0
5 1.0 3.0 11.0

Difference with previous column

>>> df.diff(axis=1)
 a b c
0 NaN 0.0 0.0
1 NaN -1.0 3.0
2 NaN -1.0 7.0
3 NaN -1.0 13.0
4 NaN 0.0 20.0
5 NaN 2.0 28.0

Difference with 3rd previous row

>>> df.diff(periods=3)
 a b c
0 NaN NaN NaN
1 NaN NaN NaN
2 NaN NaN NaN
3 3.0 2.0 15.0
4 3.0 4.0 21.0
5 3.0 6.0 27.0

Difference with following row

>>> df.diff(periods=-1)
 a b c
0 -1.0 0.0 -3.0
1 -1.0 -1.0 -5.0
2 -1.0 -1.0 -7.0
3 -1.0 -2.0 -9.0
4 -1.0 -3.0 -11.0
5 NaN NaN NaN
"""
bm_axis = self._get_block_manager_axis(axis)
self._consolidate_inplace()

if bm_axis == 0 and periods != 0:
 return self.T.diff(periods, axis=0).T

new_data = self._mgr.diff(n=periods, axis=bm_axis)
return self._constructor(new_data)

Function application

def __getitem__(


```

```

 self,
 key: Union[str, List[str]],
 ndim: int,
 subset: Optional[Union[Series, ABCDataFrame]] = None,
) -> Union[Series, ABCDataFrame]:
 """
 Sub-classes to define. Return a sliced object.

 Parameters

 key : string / list of selections
 ndim : 1,2
 requested ndim of result
 subset : object, default None
 subset to act on
 """
 if subset is None:
 subset = self
 elif subset.ndim == 1: # is Series
 return subset

 # TODO: _shallow_copy(subset)?
 return subset[key]

```

`_agg_summary_and_see_also_doc` = dedent(

The aggregation operations are always performed over an axis, either the index (default) or the column axis. This behavior is different from `numpy` aggregation functions (`mean`, `median`, `prod`, `sum`, `std`, `var`), where the default is to compute the aggregation of the flattened array, e.g., ``numpy.mean(arr\_2d)`` as opposed to ``numpy.mean(arr\_2d, axis=0)``.

`agg` is an alias for `aggregate`. Use the alias.

See Also

-----

`DataFrame.apply` : Perform any type of operations.  
`DataFrame.transform` : Perform transformation type operations.  
`core.groupby.GroupBy` : Perform operations over groups.  
`core.resample.Resampler` : Perform operations over resampled bins.  
`core.window.Rolling` : Perform operations over rolling window.  
`core.window.Expanding` : Perform operations over expanding window.  
`core.window.EWM` : Perform operation over exponential weighted window.

"""

)

`_agg_examples_doc` = dedent(

Examples

-----

```
>>> df = pd.DataFrame([[1, 2, 3],
... [4, 5, 6],
... [7, 8, 9],
... [np.nan, np.nan, np.nan]],
... columns=['A', 'B', 'C'])
```

Aggregate these functions over the rows.

```
>>> df.agg(['sum', 'min'])
 A B C
sum 12.0 15.0 18.0
```

```

min 1.0 2.0 3.0

Different aggregations per column.

>>> df.agg({'A' : ['sum', 'min'], 'B' : ['min', 'max']})
 A B
max NaN 8.0
min 1.0 2.0
sum 12.0 NaN

Aggregate over the columns.

>>> df.agg("mean", axis="columns")
0 2.0
1 5.0
2 8.0
3 NaN
dtype: float64
"""

)

@Substitution(
 see_also=_agg_summary_and_see_also_doc,
 examples=_agg_examples_doc,
 versionadded="\n.. versionadded:: 0.20.0\n",
 **_shared_doc_kwargs,
)
@Appender(_shared_docs["aggregate"])
def aggregate(self, func, axis=0, *args, **kwargs):
 axis = self._get_axis_number(axis)

 result = None
 try:
 result, how = self._aggregate(func, axis=axis, *args, **kwargs)
 except TypeError:
 pass
 if result is None:
 return self.apply(func, axis=axis, args=args, **kwargs)
 return result

def _aggregate(self, arg, axis=0, *args, **kwargs):
 if axis == 1:
 # NDFrame.aggregate returns a tuple, and we need to transpose
 # only result
 result, how = self.T._aggregate(arg, *args, **kwargs)
 result = result.T if result is not None else result
 return result, how
 return super()._aggregate(arg, *args, **kwargs)

agg = aggregate

@Appender(_shared_docs["transform"] % _shared_doc_kwargs)
def transform(self, func, axis=0, *args, **kwargs) -> "DataFrame":
 axis = self._get_axis_number(axis)
 if axis == 1:
 return self.T.transform(func, *args, **kwargs).T
 return super().transform(func, *args, **kwargs)

def apply(self, func, axis=0, raw=False, result_type=None, args=(), **kwds):
 """
 Apply a function along an axis of the DataFrame.

 Objects passed to the function are Series objects whose index is

```

either the DataFrame's index (``axis=0``) or the DataFrame's columns (``axis=1``). By default (``result\_type=None``), the final return type is inferred from the return type of the applied function. Otherwise, it depends on the `result\_type` argument.

#### Parameters

-----

`func` : function

Function to apply to each column or row.

`axis` : {0 or 'index', 1 or 'columns'}, default 0

Axis along which the function is applied:

- \* 0 or 'index': apply function to each column.
- \* 1 or 'columns': apply function to each row.

`raw` : bool, default False

Determines if row or column is passed as a Series or ndarray object:

- \* ``False`` : passes each row or column as a Series to the function.

- \* ``True`` : the passed function will receive ndarray objects instead.

If you are just applying a NumPy reduction function this will achieve much better performance.

`result_type` : {'expand', 'reduce', 'broadcast', None}, default None  
These only act when ``axis=1`` (columns):

- \* 'expand' : list-like results will be turned into columns.

- \* 'reduce' : returns a Series if possible rather than expanding list-like results. This is the opposite of 'expand'.

- \* 'broadcast' : results will be broadcast to the original shape of the DataFrame, the original index and columns will be retained.

The default behaviour (None) depends on the return value of the applied function: list-like results will be returned as a Series of those. However if the apply function returns a Series these are expanded to columns.

.. versionadded:: 0.23.0

#### args : tuple

Positional arguments to pass to `func` in addition to the array/series.

#### \*\*kwds

Additional keyword arguments to pass as keywords arguments to `func`.

#### Returns

-----

Series or DataFrame

Result of applying ``func`` along the given axis of the DataFrame.

#### See Also

-----

`DataFrame.applymap`: For elementwise operations.

`DataFrame.aggregate`: Only perform aggregating type operations.

`DataFrame.transform`: Only perform transforming type operations.

#### Examples

-----

```

>>> df = pd.DataFrame([[4, 9]] * 3, columns=['A', 'B'])
>>> df
 A B
0 4 9
1 4 9
2 4 9

Using a numpy universal function (in this case the same as
``np.sqrt(df)``):

>>> df.apply(np.sqrt)
 A B
0 2.0 3.0
1 2.0 3.0
2 2.0 3.0

Using a reducing function on either axis

>>> df.apply(np.sum, axis=0)
A 12
B 27
dtype: int64

>>> df.apply(np.sum, axis=1)
0 13
1 13
2 13
dtype: int64

Returning a list-like will result in a Series

>>> df.apply(lambda x: [1, 2], axis=1)
0 [1, 2]
1 [1, 2]
2 [1, 2]
dtype: object

Passing ``result_type='expand'`` will expand list-like results
to columns of a Dataframe

>>> df.apply(lambda x: [1, 2], axis=1, result_type='expand')
 0 1
0 1 2
1 1 2
2 1 2

Returning a Series inside the function is similar to passing
``result_type='expand'``. The resulting column names
will be the Series index.

>>> df.apply(lambda x: pd.Series([1, 2], index=['foo', 'bar']), axis=1)
 foo bar
0 1 2
1 1 2
2 1 2

Passing ``result_type='broadcast'`` will ensure the same shape
result, whether list-like or scalar is returned by the function,
and broadcast it along the axis. The resulting column names will
be the originals.

>>> df.apply(lambda x: [1, 2], axis=1, result_type='broadcast')
 A B

```

```
0 1 2
1 1 2
2 1 2
"""
from pandas.core.apply import frame_apply

op = frame_apply(
 self,
 func=func,
 axis=axis,
 raw=raw,
 result_type=result_type,
 args=args,
 kwds=kwds,
)
return op.get_result()

def applymap(self, func) -> "DataFrame":
 """
 Apply a function to a Dataframe elementwise.

 This method applies a function that accepts and returns a scalar
 to every element of a DataFrame.

 Parameters

 func : callable
 Python function, returns a single value from a single value.

 Returns

 DataFrame
 Transformed DataFrame.

 See Also

 DataFrame.apply : Apply a function along input axis of DataFrame.

 Notes

 In the current implementation applymap calls `func` twice on the
 first column/row to decide whether it can take a fast or slow
 code path. This can lead to unexpected behavior if `func` has
 side-effects, as they will take effect twice for the first
 column/row.

 Examples

 >>> df = pd.DataFrame([[1, 2.12], [3.356, 4.567]])
 >>> df
 0 1
0 1.000 2.120
1 3.356 4.567

 >>> df.applymap(lambda x: len(str(x)))
 0 1
0 3 4
1 5 5

 Note that a vectorized version of `func` often exists, which will
 be much faster. You could square each number elementwise.

 >>> df.applymap(lambda x: x**2)
```

```

 0 1
0 1.000000 4.494400
1 11.262736 20.857489

But it's better to avoid applymap in that case.

>>> df ** 2
 0 1
0 1.000000 4.494400
1 11.262736 20.857489
"""
if we have a dtype == 'M8[ns]', provide boxed values
def infer(x):
 if x.empty:
 return lib.map_infer(x, func)
 return lib.map_infer(x.astype(object).values, func)

return self.apply(infer)

Merging / joining methods

def append(
 self, other, ignore_index=False, verify_integrity=False, sort=False
) -> "DataFrame":
"""
Append rows of `other` to the end of caller, returning a new object.

Columns in `other` that are not in the caller are added as new columns.

Parameters

other : DataFrame or Series/dict-like object, or list of these
 The data to append.
ignore_index : bool, default False
 If True, do not use the index labels.
verify_integrity : bool, default False
 If True, raise ValueError on creating index with duplicates.
sort : bool, default False
 Sort columns if the columns of `self` and `other` are not aligned.

.. versionadded:: 0.23.0
.. versionchanged:: 1.0.0

 Changed to not sort by default.

Returns

DataFrame

See Also

concat : General function to concatenate DataFrame or Series objects.

Notes

If a list of dict/series is passed and the keys are all contained in
the DataFrame's index, the order of the columns in the resulting
DataFrame will be unchanged.

Iteratively appending rows to a DataFrame can be more computationally
intensive than a single concatenate. A better solution is to append
those rows to a list and then concatenate the list with the original

```

```
DataFrame all at once.
```

#### Examples

```

>>> df = pd.DataFrame([[1, 2], [3, 4]], columns=list('AB'))
>>> df
A B
0 1 2
1 3 4
>>> df2 = pd.DataFrame([[5, 6], [7, 8]], columns=list('AB'))
>>> df.append(df2)
A B
0 1 2
1 3 4
0 5 6
1 7 8
```

With `ignore\_index` set to True:

```
>>> df.append(df2, ignore_index=True)
A B
0 1 2
1 3 4
2 5 6
3 7 8
```

The following, while not recommended methods for generating DataFrames, show two ways to generate a DataFrame from multiple data sources.

Less efficient:

```
>>> df = pd.DataFrame(columns=['A'])
>>> for i in range(5):
... df = df.append({'A': i}, ignore_index=True)
>>> df
A
0 0
1 1
2 2
3 3
4 4
```

More efficient:

```
>>> pd.concat([pd.DataFrame([i], columns=['A']) for i in range(5)],
... ignore_index=True)
A
0 0
1 1
2 2
3 3
4 4
"""
if isinstance(other, (Series, dict)):
 if isinstance(other, dict):
 if not ignore_index:
 raise TypeError("Can only append a dict if ignore_index=True")
 other = Series(other)
 if other.name is None and not ignore_index:
 raise TypeError(
 "Can only append a Series if ignore_index=True "
 "or if the Series has a name"
)
```

```

index = Index([other.name], name=self.index.name)
idx_diff = other.index.difference(self.columns)
try:
 combined_columns = self.columns.append(idx_diff)
except TypeError:
 combined_columns = self.columns.astype(object).append(idx_diff)
other = (
 other.reindex(combined_columns, copy=False)
 .to_frame()
 .T.infer_objects()
 .rename_axis(index.names, copy=False)
)
if not self.columns.equals(combined_columns):
 self = self.reindex(columns=combined_columns)
elif isinstance(other, list):
 if not other:
 pass
 elif not isinstance(other[0], DataFrame):
 other = DataFrame(other)
 if (self.columns.get_indexer(other.columns) >= 0).all():
 other = other.reindex(columns=self.columns)

from pandas.core.reshape.concat import concat

if isinstance(other, (list, tuple)):
 to_concat = [self, *other]
else:
 to_concat = [self, other]
return concat(
 to_concat,
 ignore_index=ignore_index,
 verify_integrity=verify_integrity,
 sort=sort,
)

```

**def join(**

self, other, on=None, how="left", lsuffix="", rsuffix="", sort=False

**) -> "DataFrame":**

"""

Join columns of another DataFrame.

Join columns with `other` DataFrame either on index or on a key column. Efficiently join multiple DataFrame objects by index at once by passing a list.

**Parameters**

-----

**other : DataFrame, Series, or list of DataFrame**  
Index should be similar to one of the columns in this one. If a Series is passed, its name attribute must be set, and that will be used as the column name in the resulting joined DataFrame.

**on : str, list of str, or array-like, optional**  
Column or index level name(s) in the caller to join on the index in `other`, otherwise joins index-on-index. If multiple values given, the `other` DataFrame must have a MultiIndex. Can pass an array as the join key if it is not already contained in the calling DataFrame. Like an Excel VLOOKUP operation.

**how : {'left', 'right', 'outer', 'inner'}, default 'left'**  
How to handle the operation of the two objects.

- \* left: use calling frame's index (or column if on is specified)
- \* right: use `other`'s index.

```
* outer: form union of calling frame's index (or column if on is
 specified) with `other`'s index, and sort it.
 lexicographically.
* inner: form intersection of calling frame's index (or column if
 on is specified) with `other`'s index, preserving the order
 of the calling's one.
lsuffix : str, default ''
 Suffix to use from left frame's overlapping columns.
rsuffix : str, default ''
 Suffix to use from right frame's overlapping columns.
sort : bool, default False
 Order result DataFrame lexicographically by the join key. If False,
 the order of the join key depends on the join type (how keyword).
```

Returns

-----

DataFrame

A datafram containing columns from both the caller and `other`.

See Also

-----

DataFrame.merge : For column(s)-on-column(s) operations.

Notes

-----

Parameters `on`, `lsuffix`, and `rsuffix` are not supported when passing a list of `DataFrame` objects.

Support for specifying index levels as the `on` parameter was added in version 0.23.0.

Examples

-----

```
>>> df = pd.DataFrame({'key': ['K0', 'K1', 'K2', 'K3', 'K4', 'K5'],
... 'A': ['A0', 'A1', 'A2', 'A3', 'A4', 'A5']})
```

```
>>> df
```

|   | key | A  |
|---|-----|----|
| 0 | K0  | A0 |
| 1 | K1  | A1 |
| 2 | K2  | A2 |
| 3 | K3  | A3 |
| 4 | K4  | A4 |
| 5 | K5  | A5 |

```
>>> other = pd.DataFrame({'key': ['K0', 'K1', 'K2'],
... 'B': ['B0', 'B1', 'B2']})
```

```
>>> other
```

|   | key | B  |
|---|-----|----|
| 0 | K0  | B0 |
| 1 | K1  | B1 |
| 2 | K2  | B2 |

Join DataFrames using their indexes.

```
>>> df.join(other, lsuffix='_caller', rsuffix='_other')
```

|   | key_caller | A  | key_other | B   |
|---|------------|----|-----------|-----|
| 0 | K0         | A0 | K0        | B0  |
| 1 | K1         | A1 | K1        | B1  |
| 2 | K2         | A2 | K2        | B2  |
| 3 | K3         | A3 | NaN       | NaN |
| 4 | K4         | A4 | NaN       | NaN |

```
5 K5 A5 NaN NaN
```

If we want to join using the key columns, we need to set key to be the index in both `df` and `other`. The joined DataFrame will have key as its index.

```
>>> df.set_index('key').join(other.set_index('key'))
 A B
key
K0 A0 B0
K1 A1 B1
K2 A2 B2
K3 A3 NaN
K4 A4 NaN
K5 A5 NaN
```

Another option to join using the key columns is to use the `on` parameter. DataFrame.join always uses `other`'s index but we can use any column in `df`. This method preserves the original DataFrame's index in the result.

```
>>> df.join(other.set_index('key'), on='key')
 key A B
0 K0 A0 B0
1 K1 A1 B1
2 K2 A2 B2
3 K3 A3 NaN
4 K4 A4 NaN
5 K5 A5 NaN
"""
 return self._join_compat(
 other, on=on, how=how, lsuffix=lsuffix, rsuffix=rsuffix, sort=sort
)

def _join_compat(
 self, other, on=None, how="left", lsuffix="", rsuffix="", sort=False
):
 from pandas.core.reshape.merge import merge
 from pandas.core.reshape.concat import concat

 if isinstance(other, Series):
 if other.name is None:
 raise ValueError("Other Series must have a name")
 other = DataFrame({other.name: other})

 if isinstance(other, DataFrame):
 return merge(
 self,
 other,
 left_on=on,
 how=how,
 left_index=on is None,
 right_index=True,
 suffixes=(lsuffix, rsuffix),
 sort=sort,
)
 else:
 if on is not None:
 raise ValueError(
 "Joining multiple DataFrames only supported for joining on index"
)

 frames = [self] + list(other)
```

```

can_concat = all(df.index.is_unique for df in frames)

join indexes only using concat
if can_concat:
 if how == "left":
 res = concat(
 frames, axis=1, join="outer", verify_integrity=True, sort=sort
)
 return res.reindex(self.index, copy=False)
 else:
 return concat(
 frames, axis=1, join=how, verify_integrity=True, sort=sort
)

joined = frames[0]

for frame in frames[1:]:
 joined = merge(
 joined, frame, how=how, left_index=True, right_index=True
)

return joined

@Substitution("")
@Appender(_merge_doc, indents=2)
def merge(
 self,
 right,
 how="inner",
 on=None,
 left_on=None,
 right_on=None,
 left_index=False,
 right_index=False,
 sort=False,
 suffixes=("_x", "_y"),
 copy=True,
 indicator=False,
 validate=None,
) -> "DataFrame":
 from pandas.core.reshape.merge import merge

 return merge(
 self,
 right,
 how=how,
 on=on,
 left_on=left_on,
 right_on=right_on,
 left_index=left_index,
 right_index=right_index,
 sort=sort,
 suffixes=suffixes,
 copy=copy,
 indicator=indicator,
 validate=validate,
)

def round(self, decimals=0, *args, **kwargs) -> "DataFrame":
 """
 Round a DataFrame to a variable number of decimal places.

```

```
Parameters

decimals : int, dict, Series
 Number of decimal places to round each column to. If an int is
 given, round each column to the same number of places.
 Otherwise dict and Series round to variable numbers of places.
 Column names should be in the keys if `decimals` is a
 dict-like, or in the index if `decimals` is a Series. Any
 columns not included in `decimals` will be left as is. Elements
 of `decimals` which are not columns of the input will be
 ignored.
*args
 Additional keywords have no effect but might be accepted for
 compatibility with numpy.
**kwargs
 Additional keywords have no effect but might be accepted for
 compatibility with numpy.
```

#### Returns

```

DataFrame
A DataFrame with the affected columns rounded to the specified
number of decimal places.
```

#### See Also

```

numpy.around : Round a numpy array to the given number of decimals.
Series.round : Round a Series to the given number of decimals.
```

#### Examples

```

>>> df = pd.DataFrame([(0.21, 0.32), (0.01, 0.67), (0.66, 0.03), (0.21, 0.18)],
... columns=['dogs', 'cats'])
>>> df
 dogs cats
0 0.21 0.32
1 0.01 0.67
2 0.66 0.03
3 0.21 0.18
```

By providing an integer each column is rounded to the same number of decimal places

```
>>> df.round(1)
 dogs cats
0 0.2 0.3
1 0.0 0.7
2 0.7 0.0
3 0.2 0.2
```

With a dict, the number of places for specific columns can be specified with the column names as key and the number of decimal places as value

```
>>> df.round({'dogs': 1, 'cats': 0})
 dogs cats
0 0.2 0.0
1 0.0 1.0
2 0.7 0.0
3 0.2 0.0
```

Using a Series, the number of places for specific columns can be specified with the column names as index and the number of

```

decimal places as value

>>> decimals = pd.Series([0, 1], index=['cats', 'dogs'])
>>> df.round(decimals)
 dogs cats
0 0.2 0.0
1 0.0 1.0
2 0.7 0.0
3 0.2 0.0
"""
from pandas.core.reshape.concat import concat

def _dict_round(df, decimals):
 for col, vals in df.items():
 try:
 yield _series_round(vals, decimals[col])
 except KeyError:
 yield vals

def _series_round(s, decimals):
 if is_integer_dtype(s) or is_float_dtype(s):
 return s.round(decimals)
 return s

nv.validate_round(args, kwargs)

if isinstance(decimals, (dict, Series)):
 if isinstance(decimals, Series):
 if not decimals.index.is_unique:
 raise ValueError("Index of decimals must be unique")
 new_cols = list(_dict_round(self, decimals))
elif is_integer(decimals):
 # Dispatch to Series.round
 new_cols = [_series_round(v, decimals) for _, v in self.items()]
else:
 raise TypeError("decimals must be an integer, a dict-like or a Series")

if len(new_cols) > 0:
 return self._constructor(
 concat(new_cols, axis=1), index=self.index, columns=self.columns
)
else:
 return self

Statistical methods, etc.

def corr(self, method="pearson", min_periods=1) -> "DataFrame":
 """
 Compute pairwise correlation of columns, excluding NA/null values.

 Parameters

 method : {'pearson', 'kendall', 'spearman'} or callable
 Method of correlation:
 * pearson : standard correlation coefficient
 * kendall : Kendall Tau correlation coefficient
 * spearman : Spearman rank correlation
 * callable: callable with input two 1d ndarrays
 and returning a float. Note that the returned matrix from corr
 will have 1 along the diagonals and will be symmetric
 regardless of the callable's behavior.
 """

```

```
.. versionadded:: 0.24.0

min_periods : int, optional
 Minimum number of observations required per pair of columns
 to have a valid result. Currently only available for Pearson
 and Spearman correlation.

>Returns

DataFrame
 Correlation matrix.

See Also

DataFrame.corrwith : Compute pairwise correlation with another
 DataFrame or Series.
Series.corr : Compute the correlation between two Series.

Examples

>>> def histogram_intersection(a, b):
... v = np.minimum(a, b).sum().round(decimals=1)
... return v
>>> df = pd.DataFrame([(2, 3), (0, 6), (6, 0), (2, 1)],
... columns=['dogs', 'cats'])
>>> df.corr(method=histogram_intersection)
 dogs cats
dogs 1.0 0.3
cats 0.3 1.0
"""
numeric_df = self._get_numeric_data()
cols = numeric_df.columns
idx = cols.copy()
mat = numeric_df.values

if method == "pearson":
 correl = libalgos.nancorr(ensure_float64(mat), minp=min_periods)
elif method == "spearman":
 correl = libalgos.nancorr_spearman(ensure_float64(mat), minp=min_periods)
elif method == "kendall" or callable(method):
 if min_periods is None:
 min_periods = 1
 mat = ensure_float64(mat).T
 corrf = nanops.get_corr_func(method)
 K = len(cols)
 correl = np.empty((K, K), dtype=float)
 mask = np.isfinite(mat)
 for i, ac in enumerate(mat):
 for j, bc in enumerate(mat):
 if i > j:
 continue

 valid = mask[i] & mask[j]
 if valid.sum() < min_periods:
 c = np.nan
 elif i == j:
 c = 1.0
 elif not valid.all():
 c = corrf(ac[valid], bc[valid])
 else:
 c = corrf(ac, bc)
 correl[i, j] = c
```

```
 correl[j, i] = c
 else:
 raise ValueError(
 "method must be either 'pearson', "
 "'spearman', 'kendall', or a callable, "
 f"'{method}' was supplied"
)

 return self._constructor(correl, index=idx, columns=cols)

def cov(self, min_periods=None) -> "DataFrame":
 """
 Compute pairwise covariance of columns, excluding NA/null values.

 Compute the pairwise covariance among the series of a DataFrame.
 The returned data frame is the `covariance matrix`_
 <https://en.wikipedia.org/wiki/Covariance_matrix>`__ of the columns
 of the DataFrame.

 Both NA and null values are automatically excluded from the
 calculation. (See the note below about bias from missing values.)
 A threshold can be set for the minimum number of
 observations for each value created. Comparisons with observations
 below this threshold will be returned as ``NaN``.

 This method is generally used for the analysis of time series data to
 understand the relationship between different measures
 across time.

 Parameters

 min_periods : int, optional
 Minimum number of observations required per pair of columns
 to have a valid result.

 Returns

 DataFrame
 The covariance matrix of the series of the DataFrame.

 See Also

 Series.cov : Compute covariance with another Series.
 core.window.EWM.cov: Exponential weighted sample covariance.
 core.window.Expanding.cov : Expanding sample covariance.
 core.window.Rolling.cov : Rolling sample covariance.

 Notes

 Returns the covariance matrix of the DataFrame's time series.
 The covariance is normalized by N-1.

 For DataFrames that have Series that are missing data (assuming that
 data is `missing at random`_
 <https://en.wikipedia.org/wiki/Missing_data#Missing_at_random>`__)
 the returned covariance matrix will be an unbiased estimate
 of the variance and covariance between the member Series.

 However, for many applications this estimate may not be acceptable
 because the estimate covariance matrix is not guaranteed to be positive
 semi-definite. This could lead to estimate correlations having
 absolute values which are greater than one, and/or a non-invertible
 covariance matrix. See `Estimation of covariance matrices`_

```

```
<https://en.wikipedia.org/w/index.php?title=Estimation_of_covariance_matrices>`__ for more details.
```

#### Examples

-----

```
>>> df = pd.DataFrame([(1, 2), (0, 3), (2, 0), (1, 1)],
... columns=['dogs', 'cats'])
>>> df.cov()
 dogs cats
dogs 0.666667 -1.000000
cats -1.000000 1.666667

>>> np.random.seed(42)
>>> df = pd.DataFrame(np.random.randn(1000, 5),
... columns=['a', 'b', 'c', 'd', 'e'])
>>> df.cov()
 a b c d e
a 0.998438 -0.020161 0.059277 -0.008943 0.014144
b -0.020161 1.059352 -0.008543 -0.024738 0.009826
c 0.059277 -0.008543 1.010670 -0.001486 -0.000271
d -0.008943 -0.024738 -0.001486 0.921297 -0.013692
e 0.014144 0.009826 -0.000271 -0.013692 0.977795
```

\*\*Minimum number of periods\*\*

This method also supports an optional ``min\_periods`` keyword that specifies the required minimum number of non-NA observations for each column pair in order to have a valid result:

```
>>> np.random.seed(42)
>>> df = pd.DataFrame(np.random.randn(20, 3),
... columns=['a', 'b', 'c'])
>>> df.loc[df.index[:5], 'a'] = np.nan
>>> df.loc[df.index[5:10], 'b'] = np.nan
>>> df.cov(min_periods=12)
 a b c
a 0.316741 NaN -0.150812
b NaN 1.248003 0.191417
c -0.150812 0.191417 0.895202
"""

numeric_df = self._get_numeric_data()
cols = numeric_df.columns
idx = cols.copy()
mat = numeric_df.values

if notna(mat).all():
 if min_periods is not None and min_periods > len(mat):
 baseCov = np.empty((mat.shape[1], mat.shape[1]))
 baseCov.fill(np.nan)
 else:
 baseCov = np.cov(mat.T)
 baseCov = baseCov.reshape((len(cols), len(cols)))
else:
 baseCov = libalgos.nancorr(ensure_float64(mat), cov=True, minp=min_periods)

return self._constructor(baseCov, index=idx, columns=cols)

def corrwith(self, other, axis=0, drop=False, method="pearson") -> Series:
 """
 Compute pairwise correlation.

 Pairwise correlation is computed between rows or columns of
 DataFrame with rows or columns of Series or DataFrame. DataFrames
```

```
are first aligned along both axes before computing the
correlations.

Parameters

other : DataFrame, Series
 Object with which to compute correlations.
axis : {0 or 'index', 1 or 'columns'}, default 0
 The axis to use. 0 or 'index' to compute column-wise, 1 or 'columns' for
 row-wise.
drop : bool, default False
 Drop missing indices from result.
method : {'pearson', 'kendall', 'spearman'} or callable
 Method of correlation:
 * pearson : standard correlation coefficient
 * kendall : Kendall Tau correlation coefficient
 * spearman : Spearman rank correlation
 * callable: callable with input two 1d ndarrays
 and returning a float.

 .. versionadded:: 0.24.0

Returns

Series
 Pairwise correlations.

See Also

DataFrame.corr : Compute pairwise correlation of columns.
"""
axis = self._get_axis_number(axis)
this = self._get_numeric_data()

if isinstance(other, Series):
 return this.apply(lambda x: other.corr(x, method=method), axis=axis)

other = other._get_numeric_data()
left, right = this.align(other, join="inner", copy=False)

if axis == 1:
 left = left.T
 right = right.T

if method == "pearson":
 # mask missing values
 left = left + right * 0
 right = right + left * 0

 # demeaned data
 ldem = left - left.mean()
 rdm = right - right.mean()

 num = (ldem * rdm).sum()
 dom = (left.count() - 1) * left.std() * right.std()

 correl = num / dom

elif method in ["kendall", "spearman"] or callable(method):

 def c(x):
 return nanops.nancorr(x[0], x[1], method=method)
```

```

 correl = Series(
 map(c, zip(left.values.T, right.values.T)), index=left.columns
)

 else:
 raise ValueError(
 f"Invalid method {method} was passed, "
 "valid methods are: 'pearson', 'kendall', "
 "'spearmen', or callable"
)

if not drop:
 # Find non-matching labels along the given axis
 # and append missing correlations (GH 22375)
 raxis = 1 if axis == 0 else 0
 result_index = this._get_axis(raxis).union(other._get_axis(raxis))
 idx_diff = result_index.difference(correl.index)

 if len(idx_diff) > 0:
 correl = correl.append(Series([np.nan] * len(idx_diff), index=idx_diff))

return correl

ndarray-like stats methods

def count(self, axis=0, level=None, numeric_only=False):
 """
 Count non-NA cells for each column or row.

 The values `None`, `NaN`, `NaT`, and optionally `numpy.inf` (depending
 on `pandas.options.mode.use_inf_as_na`) are considered NA.

 Parameters

 axis : {0 or 'index', 1 or 'columns'}, default 0
 If 0 or 'index' counts are generated for each column.
 If 1 or 'columns' counts are generated for each row.
 level : int or str, optional
 If the axis is a `MultiIndex` (hierarchical), count along a
 particular `level`, collapsing into a `DataFrame`.
 A `str` specifies the level name.
 numeric_only : bool, default False
 Include only `float`, `int` or `boolean` data.

 Returns

 Series or DataFrame
 For each column/row the number of non-NA/null entries.
 If `level` is specified returns a `DataFrame`.

 See Also

 Series.count: Number of non-NA elements in a Series.
 DataFrame.shape: Number of DataFrame rows and columns (including NA
 elements).
 DataFrame.isna: Boolean same-sized DataFrame showing places of NA
 elements.

 Examples

 Constructing DataFrame from a dictionary:

```

```

>>> df = pd.DataFrame({"Person": ["John", "Myla", "Lewis", "John", "Myla"],
... "Age": [24., np.nan, 21., 33, 26],
... "Single": [False, True, True, True, False]})
>>> df
 Person Age Single
0 John 24.0 False
1 Myla NaN True
2 Lewis 21.0 True
3 John 33.0 True
4 Myla 26.0 False

Notice the uncounted NA values:

>>> df.count()
Person 5
Age 4
Single 5
dtype: int64

Counts for each **row**:

>>> df.count(axis='columns')
0 3
1 2
2 3
3 3
4 3
dtype: int64

Counts for one level of a `MultiIndex`:

>>> df.set_index(["Person", "Single"]).count(level="Person")
 Age
Person
John 2
Lewis 1
Myla 1
"""

axis = self._get_axis_number(axis)
if level is not None:
 return self._count_level(level, axis=axis, numeric_only=numeric_only)

if numeric_only:
 frame = self._get_numeric_data()
else:
 frame = self

GH #423
if len(frame._get_axis(axis)) == 0:
 result = Series(0, index=frame._get_agg_axis(axis))
else:
 if frame._is_mixed_type or frame._mgr.any_extension_types:
 # the or any_extension_types is really only hit for single-
 # column frames with an extension array
 result = notna(frame).sum(axis=axis)
 else:
 # GH13407
 series_counts = notna(frame).sum(axis=axis)
 counts = series_counts.values
 result = Series(counts, index=frame._get_agg_axis(axis))

```

```

 return result.astype("int64")

def _count_level(self, level, axis=0, numeric_only=False):
 if numeric_only:
 frame = self._get_numeric_data()
 else:
 frame = self

 count_axis = frame._get_axis(axis)
 agg_axis = frame._get_agg_axis(axis)

 if not isinstance(count_axis, ABCMultiIndex):
 raise TypeError(
 f"Can only count levels on hierarchical {self._get_axis_name(axis)}."
)

 # Mask NaNs: Mask rows or columns where the index level is NaN, and all
 # values in the DataFrame that are NaN
 if frame._is_mixed_type:
 # Since we have mixed types, calling notna(frame.values) might
 # upcast everything to object
 values_mask = notna(frame).values
 else:
 # But use the speedup when we have homogeneous dtypes
 values_mask = notna(frame.values)

 index_mask = notna(count_axis.get_level_values(level=level))
 if axis == 1:
 mask = index_mask & values_mask
 else:
 mask = index_mask.reshape(-1, 1) & values_mask

 if isinstance(level, str):
 level = count_axis._get_level_number(level)

 level_name = count_axis._names[level]
 level_index = count_axis.levels[level]._shallow_copy(name=level_name)
 level_codes = ensure_int64(count_axis.codes[level])
 counts = lib.count_level_2d(mask, level_codes, len(level_index), axis=axis)

 if axis == 1:
 result = DataFrame(counts, index=agg_axis, columns=level_index)
 else:
 result = DataFrame(counts, index=level_index, columns=agg_axis)

 return result

def _reduce(
 self, op, name, axis=0, skipna=True, numeric_only=None, filter_type=None, **kwds
):

 assert filter_type is None or filter_type == "bool", filter_type

 dtype_is_dt = np.array(
 [
 is_datetime64_any_dtype(values.dtype)
 for values in self._iter_column_arrays()
],
 dtype=bool,
)
 if numeric_only is None and name in ["mean", "median"] and dtype_is_dt.any():
 warnings.warn(
 "DataFrame.mean and DataFrame.median with numeric_only=None "

```

```

 "will include datetime64 and datetime64tz columns in a "
 "future version.",
 FutureWarning,
 stacklevel=3,
)
 cols = self.columns[~dtype_is_dt]
 self = self[cols]

if axis is None and filter_type == "bool":
 labels = None
 constructor = None
else:
 # TODO: Make other agg func handle axis=None properly
 axis = self._get_axis_number(axis)
 labels = self._get_agg_axis(axis)
 constructor = self._constructor

def f(x):
 return op(x, axis=axis, skipna=skipna, **kwds)

def _get_data(axis_matters):
 if filter_type is None:
 data = self._get_numeric_data()
 elif filter_type == "bool":
 if axis_matters:
 # GH#25101, GH#24434
 data = self._get_bool_data() if axis == 0 else self
 else:
 data = self._get_bool_data()
 else: # pragma: no cover
 msg = (
 f"Generating numeric_only data with filter_type {filter_type} "
 "not supported."
)
 raise NotImplementedError(msg)
 return data

if numeric_only is not None and axis in [0, 1]:
 df = self
 if numeric_only is True:
 df = _get_data(axis_matters=True)
 if axis == 1:
 df = df.T
 axis = 0

 out_dtype = "bool" if filter_type == "bool" else None

 def blk_func(values):
 if values.ndim == 1 and not isinstance(values, np.ndarray):
 # we can't pass axis=1
 return op(values, axis=0, skipna=skipna, **kwds)
 return op(values, axis=1, skipna=skipna, **kwds)

 # After possibly _get_data and transposing, we are now in the
 # simple case where we can use BlockManager._reduce
 res = df._mgr.reduce(blk_func)
 assert isinstance(res, dict)
 if len(res):
 assert len(res) == max(list(res.keys())) + 1, res.keys()
 out = df._constructor_sliced(res, index=range(len(res)), dtype=out_dtype)
 out.index = df.columns
 if axis == 0 and df.dtypes.apply(needs_i8_conversion).any():
 # FIXME: needs i8 conversion check is kludge, not sure

```

```

 # why it is necessary in this case and this case alone
 out[:] = coerce_to_dtypes(out.values, df.dtypes)
 return out

 if not self._is_homogeneous_type:
 # try to avoid self.values call

 if filter_type is None and axis == 0 and len(self) > 0:
 # operate column-wise

 # numeric_only must be None here, as other cases caught above
 # require len(self) > 0 bc frame_apply messes up empty prod/sum

 # this can end up with a non-reduction
 # but not always. if the types are mixed
 # with datelike then need to make sure a series

 # we only end up here if we have not specified
 # numeric_only and yet we have tried a
 # column-by-column reduction, where we have mixed type.
 # So let's just do what we can
 from pandas.core.apply import frame_apply

 opa = frame_apply(
 self, func=f, result_type="expand", ignore_failures=True
)
 result = opa.get_result()
 if result.ndim == self.ndim:
 result = result.iloc[0].rename(None)
 return result

 if numeric_only is None:
 data = self
 values = data.values

 try:
 result = f(values)

 except TypeError:
 # e.g. in nanops trying to convert strs to float

 # TODO: why doesnt axis matter here?
 data = _get_data(axis_matters=False)
 labels = data._get_agg_axis(axis)

 values = data.values
 with np.errstate(all="ignore"):
 result = f(values)

 else:
 if numeric_only:
 data = _get_data(axis_matters=True)
 labels = data._get_agg_axis(axis)

 values = data.values
 else:
 data = self
 values = data.values
 result = f(values)

 if filter_type == "bool" and is_object_dtype(values) and axis is None:
 # work around https://github.com/numpy/numpy/issues/10489
 # TODO: can we de-duplicate parts of this with the next block?

```

```
 result = np.bool_(result)
 elif hasattr(result, "dtype") and is_object_dtype(result.dtype):
 try:
 if filter_type is None:
 result = result.astype(np.float64)
 elif filter_type == "bool" and notna(result).all():
 result = result.astype(np.bool_)
 except (ValueError, TypeError):
 # try to coerce to the original dtypes item by item if we can
 if axis == 0:
 result = coerce_to_dtypes(result, data.dtypes)

 if constructor is not None:
 result = self._constructor_sliced(result, index=labels)
 return result

def nunique(self, axis=0, dropna=True) -> Series:
 """
 Count distinct observations over requested axis.

 Return Series with number of distinct observations. Can ignore NaN
 values.

 Parameters

 axis : {0 or 'index', 1 or 'columns'}, default 0
 The axis to use. 0 or 'index' for row-wise, 1 or 'columns' for
 column-wise.
 dropna : bool, default True
 Don't include NaN in the counts.

 Returns

 Series

 See Also

 Series.nunique: Method nunique for Series.
 DataFrame.count: Count non-NA cells for each column or row.

 Examples

 >>> df = pd.DataFrame({'A': [1, 2, 3], 'B': [1, 1, 1]})
 >>> df.nunique()
 A 3
 B 1
 dtype: int64

 >>> df.nunique(axis=1)
 0 1
 1 2
 2 2
 dtype: int64
 """
 return self.apply(Series.nunique, axis=axis, dropna=dropna)

def idxmin(self, axis=0, skipna=True) -> Series:
 """
 Return index of first occurrence of minimum over requested axis.

 NA/null values are excluded.

 Parameters
```

```

axis : {0 or 'index', 1 or 'columns'}, default 0
 The axis to use. 0 or 'index' for row-wise, 1 or 'columns' for column-wise.
skipna : bool, default True
 Exclude NA/null values. If an entire row/column is NA, the result
 will be NA.

Returns

Series
 Indexes of minima along the specified axis.

Raises

ValueError
 * If the row/column is empty

See Also

Series.idxmin : Return index of the minimum element.

Notes

This method is the DataFrame version of ``ndarray.argmin``.

Examples

Consider a dataset containing food consumption in Argentina.

>>> df = pd.DataFrame({'consumption': [10.51, 103.11, 55.48],
... 'co2_emissions': [37.2, 19.66, 1712]},
... index=['Pork', 'Wheat Products', 'Beef'])

>>> df
 consumption co2_emissions
Pork 10.51 37.20
Wheat Products 103.11 19.66
Beef 55.48 1712.00

By default, it returns the index for the minimum value in each column.

>>> df.idxmin()
consumption Pork
co2_emissions Wheat Products
dtype: object

To return the index for the minimum value in each row, use ``axis="columns"``.

>>> df.idxmin(axis="columns")
Pork consumption
Wheat Products co2_emissions
Beef consumption
dtype: object
"""
 axis = self._get_axis_number(axis)
 indices = nanops.nanargmin(self.values, axis=axis, skipna=skipna)
 index = self._get_axis(index)
 result = [index[i] if i >= 0 else np.nan for i in indices]
 return Series(result, index=self._get_agg_axis(axis))

def idxmax(self, axis=0, skipna=True) -> Series:
 """
 Return index of first occurrence of maximum over requested axis.
```

```
NA/null values are excluded.

Parameters

axis : {0 or 'index', 1 or 'columns'}, default 0
 The axis to use. 0 or 'index' for row-wise, 1 or 'columns' for column-wise.
skipna : bool, default True
 Exclude NA/null values. If an entire row/column is NA, the result
 will be NA.

Returns

Series
 Indexes of maxima along the specified axis.

Raises

ValueError
 * If the row/column is empty

See Also

Series.idxmax : Return index of the maximum element.

Notes

This method is the DataFrame version of ``ndarray.argmax``.

Examples

Consider a dataset containing food consumption in Argentina.

>>> df = pd.DataFrame({'consumption': [10.51, 103.11, 55.48],
... 'co2_emissions': [37.2, 19.66, 1712]},
... index=['Pork', 'Wheat Products', 'Beef'])

>>> df
 consumption co2_emissions
Pork 10.51 37.20
Wheat Products 103.11 19.66
Beef 55.48 1712.00

By default, it returns the index for the maximum value in each column.

>>> df.idxmax()
consumption Wheat Products
co2_emissions Beef
dtype: object

To return the index for the maximum value in each row, use ``axis="columns"``.

>>> df.idxmax(axis="columns")
Pork co2_emissions
Wheat Products consumption
Beef co2_emissions
dtype: object
"""
axis = self._get_axis_number(axis)
indices = nanops.nanargmax(self.values, axis=axis, skipna=skipna)
index = self._get_axis(index)
result = [index[i] if i >= 0 else np.nan for i in indices]
return Series(result, index=self._get_agg_axis(axis))
```

```
def _get_agg_axis(self, axis_num: int) -> Index:
 """
 Let's be explicit about this.
 """
 if axis_num == 0:
 return self.columns
 elif axis_num == 1:
 return self.index
 else:
 raise ValueError(f"Axis must be 0 or 1 (got {repr(axis_num)}))")
```

```
def mode(self, axis=0, numeric_only=False, dropna=True) -> "DataFrame":
 """
 Get the mode(s) of each element along the selected axis.
```

The mode of a set of values is the value that appears most often.  
It can be multiple values.

#### Parameters

-----

axis : {0 or 'index', 1 or 'columns'}, default 0  
The axis to iterate over while searching for the mode:

- \* 0 or 'index' : get mode of each column
- \* 1 or 'columns' : get mode of each row.

numeric\_only : bool, default False  
If True, only apply to numeric columns.  
dropna : bool, default True  
Don't consider counts of NaN/NaT.

.. versionadded:: 0.24.0

#### Returns

-----

DataFrame

The modes of each column or row.

#### See Also

-----

Series.mode : Return the highest frequency value in a Series.

Series.value\_counts : Return the counts of values in a Series.

#### Examples

-----

```
>>> df = pd.DataFrame([('bird', 2, 2),
... ('mammal', 4, np.nan),
... ('arthropod', 8, 0),
... ('bird', 2, np.nan)],
... index=('falcon', 'horse', 'spider', 'ostrich'),
... columns=('species', 'legs', 'wings'))
>>> df
 species legs wings
falcon bird 2 2.0
horse mammal 4 NaN
spider arthropod 8 0.0
ostrich bird 2 NaN
```

By default, missing values are not considered, and the mode of wings  
are both 0 and 2. The second row of species and legs contains ``NaN``,  
because they have only one mode, but the DataFrame has two rows.

```
>>> df.mode()
 species legs wings
0 bird 2.0 0.0
1 NaN NaN 2.0
```

Setting ``dropna=False`` ``NaN`` values are considered and they can be the mode (like for wings).

```
>>> df.mode(dropna=False)
 species legs wings
0 bird 2 NaN
```

Setting ``numeric\_only=True``, only the mode of numeric columns is computed, and columns of other types are ignored.

```
>>> df.mode(numeric_only=True)
 legs wings
0 2.0 0.0
1 NaN 2.0
```

To compute the mode over columns and not rows, use the axis parameter:

```
>>> df.mode(axis='columns', numeric_only=True)
 0 1
falcon 2.0 NaN
horse 4.0 NaN
spider 0.0 8.0
ostrich 2.0 NaN
"""
data = self if not numeric_only else self._get_numeric_data()

def f(s):
 return s.mode(dropna=dropna)

return data.apply(f, axis=axis)
```

```
def quantile(self, q=0.5, axis=0, numeric_only=True, interpolation="linear"):
 """
 Return values at the given quantile over requested axis.
```

#### Parameters

-----

q : float or array-like, default 0.5 (50% quantile)  
Value between 0 <= q <= 1, the quantile(s) to compute.  
axis : {0, 1, 'index', 'columns'}, default 0  
Equals 0 or 'index' for row-wise, 1 or 'columns' for column-wise.  
numeric\_only : bool, default True  
If False, the quantile of datetime and timedelta data will be  
computed as well.  
interpolation : {'linear', 'lower', 'higher', 'midpoint', 'nearest'}  
This optional parameter specifies the interpolation method to use,  
when the desired quantile lies between two data points `i` and `j`:

- \* linear: `i + (j - i) \* fraction`, where `fraction` is the  
fractional part of the index surrounded by `i` and `j`.
- \* lower: `i`.
- \* higher: `j`.
- \* nearest: `i` or `j` whichever is nearest.
- \* midpoint: (`i` + `j`) / 2.

#### Returns

-----

Series or DataFrame

```
If ``q`` is an array, a DataFrame will be returned where the
index is ``q``, the columns are the columns of self, and the
values are the quantiles.
If ``q`` is a float, a Series will be returned where the
index is the columns of self and the values are the quantiles.
```

See Also

-----  
core.window.Rolling.quantile: Rolling quantile.  
numpy.percentile: Numpy function to compute the percentile.

Examples

```
----->>> df = pd.DataFrame(np.array([[1, 1], [2, 10], [3, 100], [4, 100]]),
... columns=['a', 'b'])
>>> df.quantile(.1)
a 1.3
b 3.7
Name: 0.1, dtype: float64
>>> df.quantile([.1, .5])
 a b
0.1 1.3 3.7
0.5 2.5 55.0
```

Specifying `numeric\_only=False` will also compute the quantile of datetime and timedelta data.

```
>>> df = pd.DataFrame({'A': [1, 2],
... 'B': [pd.Timestamp('2010'),
... pd.Timestamp('2011')],
... 'C': [pd.Timedelta('1 days'),
... pd.Timedelta('2 days')]})
>>> df.quantile(0.5, numeric_only=False)
A 1.5
B 2010-07-02 12:00:00
C 1 days 12:00:00
Name: 0.5, dtype: object
"""
validate_percentile(q)

data = self._get_numeric_data() if numeric_only else self
axis = self._get_axis_number(axis)
is_transposed = axis == 1

if is_transposed:
 data = data.T

if len(data.columns) == 0:
 # GH#23925 _get_numeric_data may have dropped all columns
 cols = Index([], name=self.columns.name)
 if is_list_like(q):
 return self._constructor([], index=q, columns=cols)
 return self._constructor_sliced([], index=cols, name=q, dtype=np.float64)

result = data._mgr.quantile(
 qs=q, axis=1, interpolation=interpolation, transposed=is_transposed
)

if result.ndim == 2:
 result = self._constructor(result)
else:
 result = self._constructor_sliced(result, name=q)
```

```

 if is_transposed:
 result = result.T

 return result

def to_timestamp(
 self, freq=None, how: str = "start", axis: Axis = 0, copy: bool = True
) -> "DataFrame":
 """
 Cast to DatetimeIndex of timestamps, at *beginning* of period.

 Parameters

 freq : str, default frequency of PeriodIndex
 Desired frequency.
 how : {'s', 'e', 'start', 'end'}
 Convention for converting period to timestamp; start of period
 vs. end.
 axis : {0 or 'index', 1 or 'columns'}, default 0
 The axis to convert (the index by default).
 copy : bool, default True
 If False then underlying input data is not copied.

 Returns

 DataFrame with DatetimeIndex
 """
 new_obj = self.copy(deep=copy)

 axis_name = self._get_axis_name(axis)
 old_ax = getattr(self, axis_name)
 new_ax = old_ax.to_timestamp(freq=freq, how=how)

 setattr(new_obj, axis_name, new_ax)
 return new_obj

def to_period(self, freq=None, axis: Axis = 0, copy: bool = True) -> "DataFrame":
 """
 Convert DataFrame from DatetimeIndex to PeriodIndex.

 Convert DataFrame from DatetimeIndex to PeriodIndex with desired
 frequency (inferred from index if not passed).

 Parameters

 freq : str, default
 Frequency of the PeriodIndex.
 axis : {0 or 'index', 1 or 'columns'}, default 0
 The axis to convert (the index by default).
 copy : bool, default True
 If False then underlying input data is not copied.

 Returns

 DataFrame with PeriodIndex
 """
 new_obj = self.copy(deep=copy)

 axis_name = self._get_axis_name(axis)
 old_ax = getattr(self, axis_name)
 new_ax = old_ax.to_period(freq=freq)

```

```

setattr(new_obj, axis_name, new_ax)
return new_obj

def isin(self, values) -> "DataFrame":
 """
 Whether each element in the DataFrame is contained in values.

 Parameters

 values : iterable, Series, DataFrame or dict
 The result will only be true at a location if all the
 labels match. If `values` is a Series, that's the index. If
 `values` is a dict, the keys must be the column names,
 which must match. If `values` is a DataFrame,
 then both the index and column labels must match.

 Returns

 DataFrame
 DataFrame of booleans showing whether each element in the DataFrame
 is contained in values.

 See Also

 DataFrame.eq: Equality test for DataFrame.
 Series.isin: Equivalent method on Series.
 Series.str.contains: Test if pattern or regex is contained within a
 string of a Series or Index.

 Examples

 >>> df = pd.DataFrame({'num_legs': [2, 4], 'num_wings': [2, 0]},
 ... index=['falcon', 'dog'])
 >>> df
 num_legs num_wings
 falcon 2 2
 dog 4 0

 When ``values`` is a list check whether every value in the DataFrame
 is present in the list (which animals have 0 or 2 legs or wings)

 >>> df.isin([0, 2])
 num_legs num_wings
 falcon True True
 dog False True

 When ``values`` is a dict, we can pass values to check for each
 column separately:

 >>> df.isin({'num_wings': [0, 3]})
 num_legs num_wings
 falcon False False
 dog False True

 When ``values`` is a Series or DataFrame the index and column must
 match. Note that 'falcon' does not match based on the number of legs
 in df2.

 >>> other = pd.DataFrame({'num_legs': [8, 2], 'num_wings': [0, 2]},
 ... index=['spider', 'falcon'])
 >>> df.isin(other)
 num_legs num_wings
 falcon True True

```

```

dog False False
"""
if isinstance(values, dict):
 from pandas.core.reshape.concat import concat

 values = collections.defaultdict(list, values)
 return concat(
 (
 self.iloc[:, [i]].isin(values[col])
 for i, col in enumerate(self.columns)
),
 axis=1,
)
elif isinstance(values, Series):
 if not values.index.is_unique:
 raise ValueError("cannot compute isin with a duplicate axis.")
 return self.eq(values.reindex_like(self), axis="index")
elif isinstance(values, DataFrame):
 if not (values.columns.is_unique and values.index.is_unique):
 raise ValueError("cannot compute isin with a duplicate axis.")
 return self.eq(values.reindex_like(self))
else:
 if not is_list_like(values):
 raise TypeError(
 "only list-like or dict-like objects are allowed "
 "to be passed to DataFrame.isin(), "
 f"you passed a '{type(values).__name__}'"
)
 return DataFrame(
 algorithms.isin(self.values.ravel(), values).reshape(self.shape),
 self.index,
 self.columns,
)

Add index and columns
_AXIS_ORDERS = ["index", "columns"]
_AXIS_TO_AXIS_NUMBER: Dict[Axis, int] = {
 **NDFrame._AXIS_TO_AXIS_NUMBER,
 1: 1,
 "columns": 1,
}
_AXIS_REVERSED = True
_AXIS_LEN = len(_AXIS_ORDERS)
_info_axis_number = 1
_info_axis_name = "columns"

index: "Index" = properties.AxisProperty(
 axis=1, doc="The index (row labels) of the DataFrame."
)
columns: "Index" = properties.AxisProperty(
 axis=0, doc="The column labels of the DataFrame."
)

@property
def _AXIS_NUMBERS(self) -> Dict[str, int]:
 """.. deprecated:: 1.1.0"""
 super().___AXIS_NUMBERS
 return {"index": 0, "columns": 1}

@property
def _AXIS_NAMES(self) -> Dict[int, str]:
 """.. deprecated:: 1.1.0"""

```

```
super().___AXIS_NAMES
 return {0: "index", 1: "columns"}
```

```

Add plotting methods to DataFrame
plot = CachedAccessor("plot", pandas.plotting.PlotAccessor)
hist = pandas.plotting.hist_frame
boxplot = pandas.plotting.boxplot_frame
sparse = CachedAccessor("sparse", SparseFrameAccessor)
```

```
DataFrame._add_numeric_operations()
DataFrame._add_series_or_dataframe_operations()
```

```
ops.add_flex_arithmetic_methods(DataFrame)
ops.add_special_arithmetic_methods(DataFrame)
```

```
def _from_nested_dict(data):
 # TODO: this should be seriously cythonized
 new_data = collections.defaultdict(dict)
 for index, s in data.items():
 for col, v in s.items():
 new_data[col][index] = v
 return new_data
```

<https://github.com/pandas-dev/pandas/blob/master/pandas/io/parsers.py>

```
"""
Module contains tools for processing files into DataFrames or other objects
"""

from collections import abc, defaultdict
import csv
import datetime
from io import StringIO, TextIOWrapper
import itertools
import re
import sys
from textwrap import fill
from typing import Any, Dict, Iterable, List, Set
import warnings

import numpy as np

import pandas._libs.lib as lib
import pandas._libs.ops as libops
import pandas._libs.parsers as parsers
from pandas._libs.parsers import STR_NA_VALUES
from pandas._libs.tslibs import parsing
from pandas._typing import FilePathOrBuffer
from pandas.errors import (
 AbstractMethodError,
 EmptyDataError,
 ParserError,
 ParserWarning,
)
from pandas.util._decorators import Appender

from pandas.core.dtypes.cast import astype_nansafe
from pandas.core.dtypes.common import (
 ensure_object,
 ensure_str,
 is_bool_dtype,
 is_categorical_dtype,
 is_dict_like,
 is_dtype_equal,
 is_extension_array_dtype,
 is_file_like,
 is_float,
 is_integer,
 is_integer_dtype,
 is_list_like,
 is_object_dtype,
 is_scalar,
 is_string_dtype,
 pandas_dtype,
)
from pandas.core.dtypes.dtypes import CategoricalDtype
from pandas.core.missing import isna

from pandas.core import algorithms
from pandas.core.arrays import Categorical
from pandas.core.frame import DataFrame
from pandas.core.indexes.api import (
```

```
Index,
MultiIndex,
RangeIndex,
ensure_index_from_sequences,
)
from pandas.core.series import Series
from pandas.core.tools.datetimes import tools

from pandas.io.common import (
 get_filepath_or_buffer,
 get_handle,
 infer_compression,
 validate_header_arg,
)
from pandas.io.date_converters import generic_parser

BOM character (byte order mark)
This exists at the beginning of a file to indicate endianness
of a file (stream). Unfortunately, this marker screws up parsing,
so we need to remove it if we see it.
_BOM = "\ufeff"

_doc_read_csv_and_table = (
 r"""
{summary}

Also supports optionally iterating or breaking of the file
into chunks.

Additional help can be found in the online docs for
`IO Tools <https://pandas.pydata.org/pandas-docs/stable/user_guide/io.html>`_.

Parameters

filepath_or_buffer : str, path object or file-like object
 Any valid string path is acceptable. The string could be a URL. Valid
 URL schemes include http, ftp, s3, gs, and file. For file URLs, a host is
 expected. A local file could be: file:///localhost/path/to/table.csv.

 If you want to pass in a path object, pandas accepts any ``os.PathLike``.

 By file-like object, we refer to objects with a ``read()`` method, such as
 a file handler (e.g. via builtin ``open`` function) or ``StringIO``.

sep : str, default {_default_sep}
 Delimiter to use. If sep is None, the C engine cannot automatically detect
 the separator, but the Python parsing engine can, meaning the latter will
 be used and automatically detect the separator by Python's builtin sniffer
 tool, ``csv.Sniffer``. In addition, separators longer than 1 character and
 different from ``'\s+'`` will be interpreted as regular expressions and
 will also force the use of the Python parsing engine. Note that regex
 delimiters are prone to ignoring quoted data. Regex example: ``'\r\t'``.

delimiter : str, default ``None``
 Alias for sep.

header : int, list of int, default 'infer'
 Row number(s) to use as the column names, and the start of the
 data. Default behavior is to infer the column names: if no names
 are passed the behavior is identical to ``header=0`` and column
 names are inferred from the first line of the file, if column
 names are passed explicitly then the behavior is identical to
 ``header=None``. Explicitly pass ``header=0`` to be able to
 replace existing names. The header can be a list of integers that
 specify row locations for a multi-index on the columns
 e.g. [0,1,3]. Intervening rows that are not specified will be
```

```
skipped (e.g. 2 in this example is skipped). Note that this
parameter ignores commented lines and empty lines if
``skip_blank_lines=True``, so ``header=0`` denotes the first line of
data rather than the first line of the file.
names : array-like, optional
 List of column names to use. If the file contains a header row,
 then you should explicitly pass ``header=0`` to override the column names.
 Duplicates in this list are not allowed.
index_col : int, str, sequence of int / str, or False, default ``None``
 Column(s) to use as the row labels of the ``DataFrame``, either given as
 string name or column index. If a sequence of int / str is given, a
 MultiIndex is used.

Note: ``index_col=False`` can be used to force pandas to *not* use the first
column as the index, e.g. when you have a malformed file with delimiters at
the end of each line.
usecols : list-like or callable, optional
 Return a subset of the columns. If list-like, all elements must either
 be positional (i.e. integer indices into the document columns) or strings
 that correspond to column names provided either by the user in `names` or
 inferred from the document header row(s). For example, a valid list-like
 `usecols` parameter would be ``[0, 1, 2]`` or ``['foo', 'bar', 'baz']``.
 Element order is ignored, so ``usecols=[0, 1]`` is the same as ``[1, 0]``.
 To instantiate a DataFrame from ``data`` with element order preserved use
 ``pd.read_csv(data, usecols=['foo', 'bar'])[['foo', 'bar']]`` for columns
 in ``['foo', 'bar']`` order or
 ``pd.read_csv(data, usecols=['foo', 'bar'])[['bar', 'foo']]`` for
 ``['bar', 'foo']`` order.

If callable, the callable function will be evaluated against the column
names, returning names where the callable function evaluates to True. An
example of a valid callable argument would be ``lambda x: x.upper()`` in
['AAA', 'BBB', 'DDD']. Using this parameter results in much faster
parsing time and lower memory usage.
squeeze : bool, default False
 If the parsed data only contains one column then return a Series.
prefix : str, optional
 Prefix to add to column numbers when no header, e.g. 'X' for X0, X1, ...
mangle_dupe_cols : bool, default True
 Duplicate columns will be specified as 'X', 'X.1', ...'X.N', rather than
 'X'...'X'. Passing in False will cause data to be overwritten if there
 are duplicate names in the columns.
dtype : Type name or dict of column -> type, optional
 Data type for data or columns. E.g. {{'a': np.float64, 'b': np.int32,
 'c': 'Int64'}}}
 Use `str` or `object` together with suitable `na_values` settings
 to preserve and not interpret dtype.
 If converters are specified, they will be applied INSTEAD
 of dtype conversion.
engine : {'c', 'python'}}, optional
 Parser engine to use. The C engine is faster while the python engine is
 currently more feature-complete.
converters : dict, optional
 Dict of functions for converting values in certain columns. Keys can either
 be integers or column labels.
true_values : list, optional
 Values to consider as True.
false_values : list, optional
 Values to consider as False.
skipinitialspace : bool, default False
 Skip spaces after delimiter.
skiprows : list-like, int or callable, optional
 Line numbers to skip (0-indexed) or number of lines to skip (int)
```

at the start of the file.

If callable, the callable function will be evaluated against the row indices, returning True if the row should be skipped and False otherwise.

An example of a valid callable argument would be ``lambda x: x in [0, 2]``.

`skipfooter : int, default 0`  
 Number of lines at bottom of file to skip (Unsupported with engine='c').

`nrows : int, optional`  
 Number of rows of file to read. Useful for reading pieces of large files.

`na_values : scalar, str, list-like, or dict, optional`  
 Additional strings to recognize as NA/NaN. If dict passed, specific per-column NA values. By default the following values are interpreted as NaN: '""'  
`+ fill("", ''.join(sorted(STR_NA_VALUES)), 70, subsequent_indent="")`  
`+ """'.`

`keep_default_na : bool, default True`  
 Whether or not to include the default NaN values when parsing the data.  
 Depending on whether `na\_values` is passed in, the behavior is as follows:

- \* If `keep\_default\_na` is True, and `na\_values` are specified, `na\_values` is appended to the default NaN values used for parsing.
- \* If `keep\_default\_na` is True, and `na\_values` are not specified, only the default NaN values are used for parsing.
- \* If `keep\_default\_na` is False, and `na\_values` are specified, only the NaN values specified `na\_values` are used for parsing.
- \* If `keep\_default\_na` is False, and `na\_values` are not specified, no strings will be parsed as NaN.

Note that if `na\_filter` is passed in as False, the `keep\_default\_na` and `na\_values` parameters will be ignored.

`na_filter : bool, default True`  
 Detect missing value markers (empty strings and the value of na\_values). In data without any NAs, passing na\_filter=False can improve the performance of reading a large file.

`verbose : bool, default False`  
 Indicate number of NA values placed in non-numeric columns.

`skip_blank_lines : bool, default True`  
 If True, skip over blank lines rather than interpreting as NaN values.

`parse_dates : bool or list of int or names or list of lists or dict, \ default False`  
 The behavior is as follows:

- \* boolean. If True -> try parsing the index.
- \* list of int or names. e.g. If [1, 2, 3] -> try parsing columns 1, 2, 3 each as a separate date column.
- \* list of lists. e.g. If [[1, 3]] -> combine columns 1 and 3 and parse as a single date column.
- \* dict, e.g. {'foo' : [1, 3]} -> parse columns 1, 3 as date and call result 'foo'

If a column or index cannot be represented as an array of datetimes, say because of an unparseable value or a mixture of timezones, the column or index will be returned unaltered as an object data type. For non-standard datetime parsing, use ``pd.to\_datetime`` after ``pd.read\_csv``. To parse an index or column with a mixture of timezones, specify ``date\_parser`` to be a partially-applied :func:`pandas.to\_datetime` with ``utc=True``. See :ref:`io.csv.mixed\_timezones` for more.

Note: A fast-path exists for iso8601-formatted dates.

`infer_datetime_format : bool, default False`  
 If True and `parse\_dates` is enabled, pandas will attempt to infer the format of the datetime strings in the columns, and if it can be inferred,

```
switch to a faster method of parsing them. In some cases this can increase
the parsing speed by 5-10x.
keep_date_col : bool, default False
 If True and `parse_dates` specifies combining multiple columns then
 keep the original columns.
date_parser : function, optional
 Function to use for converting a sequence of string columns to an array of
 datetime instances. The default uses ``dateutil.parser.parser`` to do the
 conversion. Pandas will try to call `date_parser` in three different ways,
 advancing to the next if an exception occurs: 1) Pass one or more arrays
 (as defined by `parse_dates`) as arguments; 2) concatenate (row-wise) the
 string values from the columns defined by `parse_dates` into a single array
 and pass that; and 3) call `date_parser` once for each row using one or
 more strings (corresponding to the columns defined by `parse_dates`) as
 arguments.
dayfirst : bool, default False
 DD/MM format dates, international and European format.
cache_dates : bool, default True
 If True, use a cache of unique, converted dates to apply the datetime
 conversion. May produce significant speed-up when parsing duplicate
 date strings, especially ones with timezone offsets.

.. versionadded:: 0.25.0
iterator : bool, default False
 Return TextFileReader object for iteration or getting chunks with
 ``get_chunk()``.
chunksize : int, optional
 Return TextFileReader object for iteration.
 See the `IO Tools docs
<https://pandas.pydata.org/pandas-docs/stable/io.html#io-chunking>`_
 for more information on ``iterator`` and ``chunksize``.
compression : {{'infer', 'gzip', 'bz2', 'zip', 'xz', None}}, default 'infer'
 For on-the-fly decompression of on-disk data. If 'infer' and
 `filepath_or_buffer` is path-like, then detect compression from the
 following extensions: '.gz', '.bz2', '.zip', or '.xz' (otherwise no
 decompression). If using 'zip', the ZIP file must contain only one data
 file to be read in. Set to None for no decompression.
thousands : str, optional
 Thousands separator.
decimal : str, default '.'
 Character to recognize as decimal point (e.g. use ',' for European data).
lineterminator : str (length 1), optional
 Character to break file into lines. Only valid with C parser.
quotechar : str (length 1), optional
 The character used to denote the start and end of a quoted item. Quoted
 items can include the delimiter and it will be ignored.
quoting : int or csv.QUOTE_* instance, default 0
 Control field quoting behavior per ``csv.QUOTE_*`` constants. Use one of
 QUOTE_MINIMAL (0), QUOTE_ALL (1), QUOTE_NONNUMERIC (2) or QUOTE_NONE (3).
doublequote : bool, default ``True``
 When quotechar is specified and quoting is not ``QUOTE_NONE``, indicate
 whether or not to interpret two consecutive quotechar elements INSIDE a
 field as a single ``quotechar`` element.
escapechar : str (length 1), optional
 One-character string used to escape other characters.
comment : str, optional
 Indicates remainder of line should not be parsed. If found at the beginning
 of a line, the line will be ignored altogether. This parameter must be a
 single character. Like empty lines (as long as ``skip_blank_lines=True``),
 fully commented lines are ignored by the parameter `header` but not by
 `skiprows`. For example, if ``comment='#'``, parsing
 ``'#empty\\na,b,c\\n1,2,3`` with ``header=0`` will result in 'a,b,c' being
 treated as the header.
```

```
encoding : str, optional
 Encoding to use for UTF when reading/writing (ex. 'utf-8'). `List of Python
 standard encodings
 <https://docs.python.org/3/library/codecs.html#standard-encodings>`_ .
dialect : str or csv.Dialect, optional
 If provided, this parameter will override values (default or not) for the
 following parameters: `delimiter`, `doublequote`, `escapechar`,
 `skipinitialspace`, `quotechar`, and `quoting`. If it is necessary to
 override values, a ParserWarning will be issued. See csv.Dialect
 documentation for more details.
error_bad_lines : bool, default True
 Lines with too many fields (e.g. a csv line with too many commas) will by
 default cause an exception to be raised, and no DataFrame will be returned.
 If False, then these "bad lines" will be dropped from the DataFrame that is
 returned.
warn_bad_lines : bool, default True
 If error_bad_lines is False, and warn_bad_lines is True, a warning for each
 "bad line" will be output.
delim_whitespace : bool, default False
 Specifies whether or not whitespace (e.g. ``' '`` or ``'\t'``) will be
 used as the sep. Equivalent to setting ``sep='\\s+'``. If this option
 is set to True, nothing should be passed in for the ``delimiter``
 parameter.
low_memory : bool, default True
 Internally process the file in chunks, resulting in lower memory use
 while parsing, but possibly mixed type inference. To ensure no mixed
 types either set False, or specify the type with the `dtype` parameter.
 Note that the entire file is read into a single DataFrame regardless,
 use the `chunksize` or `iterator` parameter to return the data in chunks.
 (Only valid with C parser).
memory_map : bool, default False
 If a filepath is provided for `filepath_or_buffer`, map the file object
 directly onto memory and access the data directly from there. Using this
 option can improve performance because there is no longer any I/O overhead.
float_precision : str, optional
 Specifies which converter the C engine should use for floating-point
 values. The options are `None` for the ordinary converter,
 `high` for the high-precision converter, and `round_trip` for the
 round-trip converter.
```

## Returns

-----

DataFrame or TextParser

A comma-separated values (csv) file is returned as two-dimensional
data structure with labeled axes.

## See Also

-----

`DataFrame.to_csv` : Write DataFrame to a comma-separated values (csv) file.  
`read_csv` : Read a comma-separated values (csv) file into DataFrame.  
`read_fwf` : Read a table of fixed-width formatted lines into DataFrame.

## Examples

-----

```
>>> pd.{func_name}('data.csv') # doctest: +SKIP
"""
")
```

```
def _validate_integer(name, val, min_val=0):
 """
```

Checks whether the 'name' parameter for parsing is either
an integer OR float that can SAFELY be cast to an integer

```
without losing accuracy. Raises a ValueError if that is
not the case.

Parameters

name : string
 Parameter name (used for error reporting)
val : int or float
 The value to check
min_val : int
 Minimum allowed value (val < min_val will result in a ValueError)
"""
msg = f"'{name:s}' must be an integer >={min_val:d}"

if val is not None:
 if is_float(val):
 if int(val) != val:
 raise ValueError(msg)
 val = int(val)
 elif not (is_integer(val) and val >= min_val):
 raise ValueError(msg)

return val

def _validate_names(names):
 """
 Raise ValueError if the `names` parameter contains duplicates.

 Parameters

 names : array-like or None
 An array containing a list of the names used for the output DataFrame.

 Raises

 ValueError
 If names are not unique.
 """
 if names is not None:
 if len(names) != len(set(names)):
 raise ValueError("Duplicate names are not allowed.")

def _read(filepath_or_buffer: FilePathOrBuffer, kwds):
 """
 Generic reader of line files."""
 encoding = kwds.get("encoding", None)
 if encoding is not None:
 encoding = re.sub("_", "-", encoding).lower()
 kwds["encoding"] = encoding

 compression = kwds.get("compression", "infer")
 compression = infer_compression(filepath_or_buffer, compression)

 # TODO: get_filepath_or_buffer could return
 # Union[FilePathOrBuffer, s3fs.S3File, gcsfs.GCSFile]
 # though mypy handling of conditional imports is difficult.
 # See https://github.com/python/mypy/issues/1297
 fp_or_buf, _, compression, should_close = get_filepath_or_buffer(
 filepath_or_buffer, encoding, compression
)
 kwds["compression"] = compression
```

```
if kwds.get("date_parser", None) is not None:
 if isinstance(kwds["parse_dates"], bool):
 kwds["parse_dates"] = True

Extract some of the arguments (pass chunksize on).
iterator = kwds.get("iterator", False)
chunksize = _validate_integer("chunksize", kwds.get("chunksize", None), 1)
nrows = kwds.get("nrows", None)

Check for duplicates in names.
_validate_names(kwds.get("names", None))

Create the parser.
parser = TextFileReader(fp_or_buf, **kwds)

if chunksize or iterator:
 return parser

try:
 data = parser.read(nrows)
finally:
 parser.close()

if should_close:
 try:
 fp_or_buf.close()
 except ValueError:
 pass

return data

_parser_defaults = {
 "delimiter": None,
 "escapechar": None,
 "quotechar": "'",
 "quoting": csv.QUOTE_MINIMAL,
 "doublequote": True,
 "skipinitialspace": False,
 "lineterminator": None,
 "header": "infer",
 "index_col": None,
 "names": None,
 "prefix": None,
 "skiprows": None,
 "skipfooter": 0,
 "nrows": None,
 "na_values": None,
 "keep_default_na": True,
 "true_values": None,
 "false_values": None,
 "converters": None,
 "dtype": None,
 "cache_dates": True,
 "thousands": None,
 "comment": None,
 "decimal": ".",
 # 'engine': 'c',
 "parse_dates": False,
 "keep_date_col": False,
 "dayfirst": False,
 "date_parser": None,
 "usecols": None,
```

```
'iterator': False,
"chunksize": None,
"verbose": False,
"encoding": None,
"squeeze": False,
"compression": None,
"mangle_dupe_cols": True,
"infer_datetime_format": False,
"skip_blank_lines": True,
}

_c_parser_defaults = {
 "delim_whitespace": False,
 "na_filter": True,
 "low_memory": True,
 "memory_map": False,
 "error_bad_lines": True,
 "warn_bad_lines": True,
 "float_precision": None,
}

_fwf_defaults = {"colspecs": "infer", "infer_nrows": 100, "widths": None}

_c_unsupported = {"skipfooter"}
_python_unsupported = {"low_memory", "float_precision"}

_deprecated_defaults: Dict[str, Any] = {}
_deprecated_args: Set[str] = set()

def _make_parser_function(name, default_sep=","):
 def parser_f(
 filepath_or_buffer: FilePathOrBuffer,
 sep=default_sep,
 delimiter=None,
 # Column and Index Locations and Names
 header="infer",
 names=None,
 index_col=None,
 usecols=None,
 squeeze=False,
 prefix=None,
 mangle_dupe_cols=True,
 # General Parsing Configuration
 dtype=None,
 engine=None,
 converters=None,
 true_values=None,
 false_values=None,
 skipinitialspace=False,
 skiprows=None,
 skipfooter=0,
 nrows=None,
 # NA and Missing Data Handling
 na_values=None,
 keep_default_na=True,
 na_filter=True,
 verbose=False,
 skip_blank_lines=True,
 # Datetime Handling
 parse_dates=False,
 infer_datetime_format=False,
```

```

keep_date_col=False,
date_parser=None,
dayfirst=False,
cache_dates=True,
Iteration
iterator=False,
chunksize=None,
Quoting, Compression, and File Format
compression="infer",
thousands=None,
decimal: str = ".",
lineterminator=None,
quotechar="'",
quoting=csv.QUOTE_MINIMAL,
doublequote=True,
escapechar=None,
comment=None,
encoding=None,
dialect=None,
Error Handling
error_bad_lines=True,
warn_bad_lines=True,
Internal
delim_whitespace=False,
low_memory=_c_parser_defaults["low_memory"],
memory_map=False,
float_precision=None,
):
 # gh-23761
 #
 # When a dialect is passed, it overrides any of the overlapping
 # parameters passed in directly. We don't want to warn if the
 # default parameters were passed in (since it probably means
 # that the user didn't pass them in explicitly in the first place).
 #
 # "delimiter" is the annoying corner case because we alias it to
 # "sep" before doing comparison to the dialect values later on.
 # Thus, we need a flag to indicate that we need to "override"
 # the comparison to dialect values by checking if default values
 # for BOTH "delimiter" and "sep" were provided.
 if dialect is not None:
 sep_override = delimiter is None and sep == default_sep
 kwds = dict(sep_override=sep_override)
 else:
 kwds = dict()

 # Alias sep -> delimiter.
 if delimiter is None:
 delimiter = sep

 if delim_whitespace and delimiter != default_sep:
 raise ValueError(
 "Specified a delimiter with both sep and "
 "'delim_whitespace=True'; you can only specify one."
)

 if engine is not None:
 engine_specified = True
 else:
 engine = "c"
 engine_specified = False

```

```
kwds.update(
 delimiter=delimiter,
 engine=engine,
 dialect=dialect,
 compression=compression,
 engine_specified=engine_specified,
 doublequote=doublequote,
 escapechar=escapechar,
 quotechar=quotechar,
 quoting=quoting,
 skipinitialspace=skipinitialspace,
 lineterminator=lineterminator,
 header=header,
 index_col=index_col,
 names=names,
 prefix=prefix,
 skiprows=skiprows,
 skipfooter=skipfooter,
 na_values=na_values,
 true_values=true_values,
 false_values=false_values,
 keep_default_na=keep_default_na,
 thousands=thousands,
 comment=comment,
 decimal=decimal,
 parse_dates=parse_dates,
 keep_date_col=keep_date_col,
 dayfirst=dayfirst,
 date_parser=date_parser,
 cache_dates=cache_dates,
 nrows=nrows,
 iterator=iterator,
 chunksize=chunksize,
 converters=converters,
 dtype=dtype,
 usecols=usecols,
 verbose=verbose,
 encoding=encoding,
 squeeze=squeeze,
 memory_map=memory_map,
 float_precision=float_precision,
 na_filter=na_filter,
 delim_whitespace=delim_whitespace,
 warn_bad_lines=warn_bad_lines,
 error_bad_lines=error_bad_lines,
 low_memory=low_memory,
 mangle_dupe_cols=mangle_dupe_cols,
 infer_datetime_format=infer_datetime_format,
 skip_blank_lines=skip_blank_lines,
)
return _read(filepath_or_buffer, kwds)

parser_f.__name__ = name

return parser_f

read_csv = _make_parser_function("read_csv", default_sep=",")
read_csv = Appender(
 _doc_read_csv_and_table.format(
 func_name="read_csv",
 summary="Read a comma-separated values (csv) file into DataFrame.",
)
)
```

```
 _default_sep=" ', ''",
)
) (read_csv)

read_table = _make_parser_function("read_table", default_sep="\t")
read_table = Appender(
 _doc_read_csv_and_table.format(
 func_name="read_table",
 summary="Read general delimited file into DataFrame.",
 _default_sep=r'\'\t' (tab-stop)",
)
) (read_table)

def read_fwf(
 filepath_or_buffer: FilePathOrBuffer,
 colspecs="infer",
 widths=None,
 infer_nrows=100,
 **kwds,
):
 """
 Read a table of fixed-width formatted lines into DataFrame.

 Also supports optionally iterating or breaking of the file
 into chunks.

 Additional help can be found in the `online docs for IO Tools
 <https://pandas.pydata.org/pandas-docs/stable/user_guide/io.html>`_.

 Parameters

 filepath_or_buffer : str, path object or file-like object
 Any valid string path is acceptable. The string could be a URL. Valid
 URL schemes include http, ftp, s3, and file. For file URLs, a host is
 expected. A local file could be:
 ``file://localhost/path/to/table.csv``.

 If you want to pass in a path object, pandas accepts any
 ``os.PathLike``.

 By file-like object, we refer to objects with a ``read()`` method,
 such as a file handler (e.g. via builtin ``open`` function)
 or ``StringIO``.

 colspecs : list of tuple (int, int) or 'infer'. optional
 A list of tuples giving the extents of the fixed-width
 fields of each line as half-open intervals (i.e., [from, to[)).
 String value 'infer' can be used to instruct the parser to try
 detecting the column specifications from the first 100 rows of
 the data which are not being skipped via skiprows (default='infer').

 widths : list of int, optional
 A list of field widths which can be used instead of 'colspecs' if
 the intervals are contiguous.

 infer_nrows : int, default 100
 The number of rows to consider when letting the parser determine the
 'colspecs'.

 .. versionadded:: 0.24.0

 **kwds : optional
 Optional keyword arguments can be passed to ``TextFileReader``.

 Returns

```

```
DataFrame or TextParser
A comma-separated values (csv) file is returned as two-dimensional
data structure with labeled axes.
```

```
See Also
```

```

```

```
DataFrame.to_csv : Write DataFrame to a comma-separated values (csv) file.
read_csv : Read a comma-separated values (csv) file into DataFrame.
```

```
Examples
```

```

```

```
>>> pd.read_fwf('data.csv') # doctest: +SKIP
"""
Check input arguments.
if colspecs is None and widths is None:
 raise ValueError("Must specify either colspecs or widths")
elif colspecs not in (None, "infer") and widths is not None:
 raise ValueError("You must specify only one of 'widths' and 'colspecs'")

Compute 'colspecs' from 'widths', if specified.
if widths is not None:
 colspecs, col = [], 0
 for w in widths:
 colspecs.append((col, col + w))
 col += w

kwds["colspecs"] = colspecs
kwds["infer_nrows"] = infer_nrows
kwds["engine"] = "python-fwf"
return _read(filepath_or_buffer, kwds)
```

```
class TextFileReader(abc.Iterator):
```

```
"""
```

```
Passed dialect overrides any of the related parser options
```

```
"""

```

```
def __init__(self, f, engine=None, **kwds):
 self.f = f

 if engine is not None:
 engine_specified = True
 else:
 engine = "python"
 engine_specified = False

 self._engine_specified = kwds.get("engine_specified", engine_specified)

 if kwds.get("dialect") is not None:
 dialect = kwds["dialect"]
 if dialect in csv.list_dialects():
 dialect = csv.get_dialect(dialect)

 # Any valid dialect should have these attributes.
 # If any are missing, we will raise automatically.
 for param in (
 "delimiter",
 "doublequote",
 "escapechar",
 "skipinitialspace",
```

```

 "quoteflag",
 "quoting",
):

 try:
 dialect_val = getattr(dialect, param)
 except AttributeError as err:
 raise ValueError(
 f"Invalid dialect {kwds['dialect']} provided"
) from err
 parser_default = _parser_defaults[param]
 provided = kwds.get(param, parser_default)

 # Messages for conflicting values between the dialect
 # instance and the actual parameters provided.
 conflict_msgs = []

 # Don't warn if the default parameter was passed in,
 # even if it conflicts with the dialect (gh-23761).
 if provided != parser_default and provided != dialect_val:
 msg = (
 f"Conflicting values for '{param}': '{provided}' was "
 f"'provided', but the dialect specifies '{dialect_val}'. "
 "Using the dialect-specified value."
)

 # Annoying corner case for not warning about
 # conflicts between dialect and delimiter parameter.
 # Refer to the outer "_read_" function for more info.
 if not (param == "delimiter" and kwds.pop("sep_override", False)):
 conflict_msgs.append(msg)

 if conflict_msgs:
 warnings.warn(
 "\n\n".join(conflict_msgs), ParserWarning, stacklevel=2
)
 kwds[param] = dialect_val

 if kwds.get("skipfooter"):
 if kwds.get("iterator") or kwds.get("chunksize"):
 raise ValueError("'skipfooter' not supported for 'iteration'")
 if kwds.get("nrows"):
 raise ValueError("'skipfooter' not supported with 'nrows'")

 if kwds.get("header", "infer") == "infer":
 kwds["header"] = 0 if kwds.get("names") is None else None

 self.orig_options = kwds

 # miscellanea
 self.engine = engine
 self._engine = None
 self._currow = 0

 options = self._get_options_with_defaults(engine)

 self.chunksize = options.pop("chunksize", None)
 self.nrows = options.pop("nrows", None)
 self.squeeze = options.pop("squeeze", False)

 # might mutate self.engine
 self.engine = self._check_file_or_buffer(f, engine)
 self.options, self.engine = self._clean_options(options, engine)

```

```

 if "has_index_names" in kwds:
 self.options["has_index_names"] = kwds["has_index_names"]

 self._make_engine(self.engine)

 def close(self):
 self._engine.close()

 def _get_options_with_defaults(self, engine):
 kwds = self.orig_options

 options = {}

 for argname, default in _parser_defaults.items():
 value = kwds.get(argname, default)

 # see gh-12935
 if argname == "mangle_dupe_cols" and not value:
 raise ValueError("Setting mangle_dupe_cols=False is not supported yet")
 else:
 options[argname] = value

 for argname, default in _c_parser_defaults.items():
 if argname in kwds:
 value = kwds[argname]

 if engine != "c" and value != default:
 if "python" in engine and argname not in _python_unsupported:
 pass
 elif value == _deprecated_defaults.get(argname, default):
 pass
 else:
 raise ValueError(
 f"The {repr(argname)} option is not supported with the "
 f"{repr(engine)} engine"
)
 else:
 value = _deprecated_defaults.get(argname, default)
 options[argname] = value

 if engine == "python-fwf":
 for argname, default in _fwf_defaults.items():
 options[argname] = kwds.get(argname, default)

 return options

 def _check_file_or_buffer(self, f, engine):
 # see gh-16530
 if is_file_like(f):
 next_attr = "__next__"

 # The C engine doesn't need the file-like to have the "next" or
 # "__next__" attribute. However, the Python engine explicitly calls
 # "next(...)" when iterating through such an object, meaning it
 # needs to have that attribute ("next" for Python 2.x, "__next__"
 # for Python 3.x)
 if engine != "c" and not hasattr(f, next_attr):
 msg = "The 'python' engine cannot iterate through this file buffer."
 raise ValueError(msg)

 return engine

 def _clean_options(self, options, engine):

```

```

result = options.copy()

engine_specified = self._engine_specified
fallback_reason = None

sep = options["delimiter"]
delim_whitespace = options["delim_whitespace"]

C engine not supported yet
if engine == "c":
 if options["skipfooter"] > 0:
 fallback_reason = "the 'c' engine does not support skipfooter"
 engine = "python"

encoding = sys.getfilesystemencoding() or "utf-8"
if sep is None and not delim_whitespace:
 if engine == "c":
 fallback_reason = (
 "the 'c' engine does not support "
 "sep=None with delim_whitespace=False"
)
 engine = "python"
elif sep is not None and len(sep) > 1:
 if engine == "c" and sep == r"\s+":
 result["delim_whitespace"] = True
 del result["delimiter"]
 elif engine not in ("python", "python-fwf"):
 # wait until regex engine integrated
 fallback_reason = (
 "the 'c' engine does not support "
 "regex separators (separators > 1 char and "
 "r'different from '\s+' are interpreted as regex)"
)
 engine = "python"
 elif delim_whitespace:
 if "python" in engine:
 result["delimiter"] = r"\s+"
elif sep is not None:
 encodeable = True
 try:
 if len(sep.encode(encoding)) > 1:
 encodeable = False
 except UnicodeDecodeError:
 encodeable = False
 if not encodeable and engine not in ("python", "python-fwf"):
 fallback_reason = (
 f"the separator encoded in {encoding} "
 "is > 1 char long, and the 'c' engine "
 "does not support such separators"
)
 engine = "python"

quotechar = options["quotechar"]
if quotechar is not None and isinstance(quotechar, (str, bytes)):
 if (
 len(quotechar) == 1
 and ord(quotechar) > 127
 and engine not in ("python", "python-fwf")
):
 fallback_reason = (
 "ord(quotechar) > 127, meaning the "
 "quotechar is larger than one byte, "
 "and the 'c' engine does not support such quotechars"
)

```

```
)
 engine = "python"

if fallback_reason and engine_specified:
 raise ValueError(fallback_reason)

if engine == "c":
 for arg in _c_unsupported:
 del result[arg]

if "python" in engine:
 for arg in _python_unsupported:
 if fallback_reason and result[arg] != _c_parser_defaults[arg]:
 raise ValueError(
 "Falling back to the 'python' engine because "
 f"{fallback_reason}, but this causes {repr(arg)} to be "
 "ignored as it is not supported by the 'python' engine."
)
 del result[arg]

if fallback_reason:
 warnings.warn(
 (
 "Falling back to the 'python' engine because "
 f"{fallback_reason}; you can avoid this warning by specifying "
 "engine='python'.
),
 ParserWarning,
 stacklevel=5,
)

index_col = options["index_col"]
names = options["names"]
converters = options["converters"]
na_values = options["na_values"]
skiprows = options["skiprows"]

validate_header_arg(options["header"])

depr_warning = ""

for arg in _deprecated_args:
 parser_default = _c_parser_defaults[arg]
 depr_default = _deprecated_defaults[arg]

 msg = (
 f"The {repr(arg)} argument has been deprecated and will be "
 "removed in a future version."
)

 if result.get(arg, depr_default) != depr_default:
 depr_warning += msg + "\n\n"
 else:
 result[arg] = parser_default

if depr_warning != "":
 warnings.warn(depr_warning, FutureWarning, stacklevel=2)

if index_col is True:
 raise ValueError("The value of index_col couldn't be 'True'")
if _is_index_col(index_col):
 if not isinstance(index_col, (list, tuple, np.ndarray)):
 index_col = [index_col]
```

```

result["index_col"] = index_col

names = list(names) if names is not None else names

type conversion-related
if converters is not None:
 if not isinstance(converters, dict):
 raise TypeError(
 "Type converters must be a dict or subclass, "
 f"input was a {type(converters).__name__}"
)
else:
 converters = {}

Converting values to NA
keep_default_na = options["keep_default_na"]
na_values, na_fvalues = _clean_na_values(na_values, keep_default_na)

handle skiprows; this is internally handled by the
c-engine, so only need for python parsers
if engine != "c":
 if is_integer(skiprows):
 skiprows = list(range(skiprows))
 if skiprows is None:
 skiprows = set()
 elif not callable(skiprows):
 skiprows = set(skiprows)

put stuff back
result["names"] = names
result["converters"] = converters
result["na_values"] = na_values
result["na_fvalues"] = na_fvalues
result["skiprows"] = skiprows

return result, engine

def __next__(self):
 try:
 return self.get_chunk()
 except StopIteration:
 self.close()
 raise

def _make_engine(self, engine="c"):
 if engine == "c":
 self._engine = CParserWrapper(self.f, **self.options)
 else:
 if engine == "python":
 klass = PythonParser
 elif engine == "python-fwf":
 klass = FixedWidthFieldParser
 else:
 raise ValueError(
 f"Unknown engine: {engine} (valid options "
 "'are \"c\", \"python\", or \"python-fwf\")"
)
 self._engine = klass(self.f, **self.options)

def _failover_to_python(self):
 raise AbstractMethodError(self)

def read(self, nrows=None):

```

```

nrows = _validate_integer("nrows", nrows)
ret = self._engine.read(nrows)

May alter columns / col_dict
index, columns, col_dict = self._create_index(ret)

if index is None:
 if col_dict:
 # Any column is actually fine:
 new_rows = len(next(iter(col_dict.values())))
 index = RangeIndex(self._currow, self._currow + new_rows)
 else:
 new_rows = 0
else:
 new_rows = len(index)

df = DataFrame(col_dict, columns=columns, index=index)

self._currow += new_rows

if self.squeeze and len(df.columns) == 1:
 return df[df.columns[0]].copy()
return df

def _create_index(self, ret):
 index, columns, col_dict = ret
 return index, columns, col_dict

def get_chunk(self, size=None):
 if size is None:
 size = self.chunksize
 if self.nrows is not None:
 if self._currow >= self.nrows:
 raise StopIteration
 size = min(size, self.nrows - self._currow)
 return self.read(nrows=size)

def _is_index_col(col):
 return col is not None and col is not False

def _is_potential_multi_index(columns):
 """
 Check whether or not the `columns` parameter
 could be converted into a MultiIndex.

 Parameters

 columns : array-like
 Object which may or may not be convertible into a MultiIndex

 Returns

 boolean : Whether or not columns could become a MultiIndex
 """
 return (
 len(columns)
 and not isinstance(columns, MultiIndex)
 and all(isinstance(c, tuple) for c in columns)
)

```

```
def _evaluate_usecols(usecols, names):
 """
 Check whether or not the 'usecols' parameter
 is a callable. If so, enumerates the 'names'
 parameter and returns a set of indices for
 each entry in 'names' that evaluates to True.
 If not a callable, returns 'usecols'.
 """
 if callable(usecols):
 return {i for i, name in enumerate(names) if usecols(name)}
 return usecols

def _validate_usecols_names(usecols, names):
 """
 Validates that all usecols are present in a given
 list of names. If not, raise a ValueError that
 shows what usecols are missing.

 Parameters

 usecols : iterable of usecols
 The columns to validate are present in names.
 names : iterable of names
 The column names to check against.

 Returns

 usecols : iterable of usecols
 The `usecols` parameter if the validation succeeds.

 Raises

 ValueError : Columns were missing. Error message will list them.
 """
 missing = [c for c in usecols if c not in names]
 if len(missing) > 0:
 raise ValueError(
 f"Usecols do not match columns, columns expected but not found: {missing}"
)

 return usecols

def _validate_skipfooter_arg(skipfooter):
 """
 Validate the 'skipfooter' parameter.

 Checks whether 'skipfooter' is a non-negative integer.
 Raises a ValueError if that is not the case.

 Parameters

 skipfooter : non-negative integer
 The number of rows to skip at the end of the file.

 Returns

 validated_skipfooter : non-negative integer
 The original input if the validation succeeds.

 Raises

```

```

ValueError : 'skipfooter' was not a non-negative integer.
"""
if not is_integer(skipfooter):
 raise ValueError("skipfooter must be an integer")

if skipfooter < 0:
 raise ValueError("skipfooter cannot be negative")

return skipfooter

def _validate_usecols_arg(usecols):
 """
 Validate the 'usecols' parameter.

 Checks whether or not the 'usecols' parameter contains all integers
 (column selection by index), strings (column by name) or is a callable.
 Raises a ValueError if that is not the case.

 Parameters

 usecols : list-like, callable, or None
 List of columns to use when parsing or a callable that can be used
 to filter a list of table columns.

 Returns

 usecols_tuple : tuple
 A tuple of (verified_usecols, usecols_dtype).

 'verified_usecols' is either a set if an array-like is passed in or
 'usecols' if a callable or None is passed in.

 'usecols_dtype` is the inferred dtype of 'usecols' if an array-like
 is passed in or None if a callable or None is passed in.
 """
 msg = (
 "'usecols' must either be list-like of all strings, all unicode, "
 "all integers or a callable."
)
 if usecols is not None:
 if callable(usecols):
 return usecols, None

 if not is_list_like(usecols):
 # see gh-20529
 #
 # Ensure it is iterable container but not string.
 raise ValueError(msg)

 usecols_dtype = lib.infer_dtype(usecols, skipna=False)

 if usecols_dtype not in ("empty", "integer", "string"):
 raise ValueError(msg)

 usecols = set(usecols)

 return usecols, usecols_dtype
return usecols, None

def _validate_parse_dates_arg(parse_dates):
 """

```

```

Check whether or not the 'parse_dates' parameter
is a non-boolean scalar. Raises a ValueError if
that is the case.
"""
msg = (
 "Only booleans, lists, and dictionaries are accepted "
 "for the 'parse_dates' parameter"
)

if parse_dates is not None:
 if is_scalar(parse_dates):
 if not lib.is_bool(parse_dates):
 raise TypeError(msg)

 elif not isinstance(parse_dates, (list, dict)):
 raise TypeError(msg)

return parse_dates

class ParserBase:
 def __init__(self, kwds):
 self.names = kwds.get("names")
 self.orig_names = None
 self.prefix = kwds.pop("prefix", None)

 self.index_col = kwds.get("index_col", None)
 self.unnamed_cols = set()
 self.index_names = None
 self.col_names = None

 self.parse_dates = _validate_parse_dates_arg(kwds.pop("parse_dates", False))
 self.date_parser = kwds.pop("date_parser", None)
 self.dayfirst = kwds.pop("dayfirst", False)
 self.keep_date_col = kwds.pop("keep_date_col", False)

 self.na_values = kwds.get("na_values")
 self.na_fvalues = kwds.get("na_fvalues")
 self.na_filter = kwds.get("na_filter", False)
 self.keep_default_na = kwds.get("keep_default_na", True)

 self.true_values = kwds.get("true_values")
 self.false_values = kwds.get("false_values")
 self.mangle_dupe_cols = kwds.get("mangle_dupe_cols", True)
 self.infer_datetime_format = kwds.pop("infer_datetime_format", False)
 self.cache_dates = kwds.pop("cache_dates", True)

 self._date_conv = _make_date_converter(
 date_parser=self.date_parser,
 dayfirst=self.dayfirst,
 infer_datetime_format=self.infer_datetime_format,
 cache_dates=self.cache_dates,
)

 # validate header options for mi
 self.header = kwds.get("header")
 if isinstance(self.header, (list, tuple, np.ndarray)):
 if not all(map(is_integer, self.header)):
 raise ValueError("header must be integer or list of integers")
 if any(i < 0 for i in self.header):
 raise ValueError(
 "cannot specify multi-index header with negative integers"
)

```

```

 if kwds.get("usecols"):
 raise ValueError(
 "cannot specify usecols when specifying a multi-index header"
)
 if kwds.get("names"):
 raise ValueError(
 "cannot specify names when specifying a multi-index header"
)

 # validate index_col that only contains integers
 if self.index_col is not None:
 is_sequence = isinstance(self.index_col, (list, tuple, np.ndarray))
 if not (
 is_sequence
 and all(map(is_integer, self.index_col))
 or is_integer(self.index_col)
):
 raise ValueError(
 "index_col must only contain row numbers "
 "when specifying a multi-index header"
)
 elif self.header is not None:
 # GH 27394
 if self.prefix is not None:
 raise ValueError(
 "Argument prefix must be None if argument header is not None"
)
 # GH 16338
 elif not is_integer(self.header):
 raise ValueError("header must be integer or list of integers")
 # GH 27779
 elif self.header < 0:
 raise ValueError(
 "Passing negative integer to header is invalid. "
 "For no header, use header=None instead"
)

 self._name_processed = False

 self._first_chunk = True

 # GH 13932
 # keep references to file handles opened by the parser itself
 self.handles = []

def _validate_parse_dates_presence(self, columns: List[str]) -> None:
 """
 Check if parse_dates are in columns.

 If user has provided names for parse_dates, check if those columns
 are available.

 Parameters

 columns : list
 List of names of the dataframe.

 Raises

 ValueError
 If column to parse_date is not in dataframe.
 """

```

```

cols_needed: Iterable
if is_dict_like(self.parse_dates):
 cols_needed = itertools.chain(*self.parse_dates.values())
elif is_list_like(self.parse_dates):
 # a column in parse_dates could be represented
 # ColReference = Union[int, str]
 # DateGroups = List[ColReference]
 # ParseDates = Union[DateGroups, List[DateGroups],
 # Dict[ColReference, DateGroups]]
 cols_needed = itertools.chain.from_iterable(
 col if is_list_like(col) else [col] for col in self.parse_dates
)
else:
 cols_needed = []

get only columns that are references using names (str), not by index
missing_cols = ", ".join(
 sorted(
 {
 col
 for col in cols_needed
 if isinstance(col, str) and col not in columns
 }
)
)
if missing_cols:
 raise ValueError(
 f"Missing column provided to 'parse_dates': '{missing_cols}'"
)

def close(self):
 for f in self.handles:
 f.close()

@property
def _has_complex_date_col(self):
 return isinstance(self.parse_dates, dict) or (
 isinstance(self.parse_dates, list)
 and len(self.parse_dates) > 0
 and isinstance(self.parse_dates[0], list)
)

def _should_parse_dates(self, i):
 if isinstance(self.parse_dates, bool):
 return self.parse_dates
 else:
 if self.index_names is not None:
 name = self.index_names[i]
 else:
 name = None
 j = self.index_col[i]

 if is_scalar(self.parse_dates):
 return (j == self.parse_dates) or (
 name is not None and name == self.parse_dates
)
 else:
 return (j in self.parse_dates) or (
 name is not None and name in self.parse_dates
)

def _extract_multi_indexer_columns(
 self, header, index_names, col_names, passed_names=False
):
 """
 Extracts columns from the header and index names based on the passed
 names. If passed_names is False, it uses the col_names. Otherwise, it
 uses the index_names. If col_names is None, it uses the index_names.
 If index_names is None, it uses the col_names. If both are None, it
 uses the header.
 """
 if passed_names:
 names = index_names
 else:
 names = col_names
 if names is None:
 names = header
 if len(names) < len(index_names):
 names += index_names[-len(names):]
 if len(names) < len(col_names):
 names += col_names[-len(names):]
 return names

```

```

) :
 """
 extract and return the names, index_names, col_names
 header is a list-of-lists returned from the parsers
 """
 if len(header) < 2:
 return header[0], index_names, col_names, passed_names

 # the names are the tuples of the header that are not the index cols
 # 0 is the name of the index, assuming index_col is a list of column
 # numbers
 ic = self.index_col
 if ic is None:
 ic = []

 if not isinstance(ic, (list, tuple, np.ndarray)):
 ic = [ic]
 sic = set(ic)

 # clean the index_names
 index_names = header.pop(-1)
 index_names, names, index_col = _clean_index_names(
 index_names, self.index_col, self.unnamed_cols
)

 # extract the columns
 field_count = len(header[0])

 def extract(r):
 return tuple(r[i] for i in range(field_count) if i not in sic)

 columns = list(zip(*(extract(r) for r in header)))
 names = ic + columns

 # If we find unnamed columns all in a single
 # level, then our header was too long.
 for n in range(len(columns[0])):
 if all(ensure_str(col[n]) in self.unnamed_cols for col in columns):
 header = ",".join(str(x) for x in self.header)
 raise ParserError(
 f"Passed header=[{header}] are too many rows "
 "for this multi_index of columns"
)

 # Clean the column names (if we have an index_col).
 if len(ic):
 col_names = [
 r[0] if (len(r[0])) and r[0] not in self.unnamed_cols else None
 for r in header
]
 else:
 col_names = [None] * len(header)

 passed_names = True

 return names, index_names, col_names, passed_names

def _maybe_dedup_names(self, names):
 # see gh-7160 and gh-9424: this helps to provide
 # immediate alleviation of the duplicate names
 # issue and appears to be satisfactory to users,
 # but ultimately, not needing to butcher the names
 # would be nice!

```

```

if self.mangle_dupe_cols:
 names = list(names) # so we can index
 counts = defaultdict(int)
 is_potential_mi = _is_potential_multi_index(names)

 for i, col in enumerate(names):
 cur_count = counts[col]

 while cur_count > 0:
 counts[col] = cur_count + 1

 if is_potential_mi:
 col = col[:-1] + (f"{col[-1]}.{cur_count}",)
 else:
 col = f"{col}.{cur_count}"
 cur_count = counts[col]

 names[i] = col
 counts[col] = cur_count + 1

return names

def _maybe_make_multi_index_columns(self, columns, col_names=None):
 # possibly create a column mi here
 if _is_potential_multi_index(columns):
 columns = MultiIndex.from_tuples(columns, names=col_names)
 return columns

def _make_index(self, data, alldata, columns, indexnamerow=False):
 if not _is_index_col(self.index_col) or not self.index_col:
 index = None

 elif not self._has_complex_date_col:
 index = self._get_simple_index(alldata, columns)
 index = self._agg_index(index)
 elif self._has_complex_date_col:
 if not self._name_processed:
 (self.index_names, _, self.index_col) = _clean_index_names(
 list(columns), self.index_col, self.unnamed_cols
)
 self._name_processed = True
 index = self._get_complex_date_index(data, columns)
 index = self._agg_index(index, try_parse_dates=False)

 # add names for the index
 if indexnamerow:
 coffset = len(indexnamerow) - len(columns)
 index = index.set_names(indexnamerow[:coffset])

 # maybe create a mi on the columns
 columns = self._maybe_make_multi_index_columns(columns, self.col_names)

 return index, columns

_implicit_index = False

def _get_simple_index(self, data, columns):
 def ix(col):
 if not isinstance(col, str):
 return col
 raise ValueError(f"Index {col} invalid")

 to_remove = []

```

```

index = []
for idx in self.index_col:
 i = ix(idx)
 to_remove.append(i)
 index.append(data[i])

remove index items from content and columns, don't pop in
loop
for i in sorted(to_remove, reverse=True):
 data.pop(i)
 if not self._implicit_index:
 columns.pop(i)

return index

def _get_complex_date_index(self, data, col_names):
 def _get_name(icol):
 if isinstance(icol, str):
 return icol

 if col_names is None:
 raise ValueError(f"Must supply column order to use {icol!s} as index")

 for i, c in enumerate(col_names):
 if i == icol:
 return c

 to_remove = []
 index = []
 for idx in self.index_col:
 name = _get_name(idx)
 to_remove.append(name)
 index.append(data[name])

 # remove index items from content and columns, don't pop in
 # loop
 for c in sorted(to_remove, reverse=True):
 data.pop(c)
 col_names.remove(c)

 return index

def _agg_index(self, index, try_parse_dates=True):
 arrays = []

 for i, arr in enumerate(index):

 if try_parse_dates and self._should_parse_dates(i):
 arr = self._date_conv(arr)

 if self.na_filter:
 col_na_values = self.na_values
 col_na_fvalues = self.na_fvalues
 else:
 col_na_values = set()
 col_na_fvalues = set()

 if isinstance(self.na_values, dict):
 col_name = self.index_names[i]
 if col_name is not None:
 col_na_values, col_na_fvalues = _get_na_values(
 col_name, self.na_values, self.na_fvalues, self.keep_default_na
)

```

```

 arr, _ = self._infer_types(arr, col_na_values | col_na_fvalues)
 arrays.append(arr)

 names = self.index_names
 index = ensure_index_from_sequences(arrays, names)

 return index

def _convert_to_ndarrays(
 self, dct, na_values, na_fvalues, verbose=False, converters=None, dtypes=None
):
 result = {}
 for c, values in dct.items():
 conv_f = None if converters is None else converters.get(c, None)
 if isinstance(dtypes, dict):
 cast_type = dtypes.get(c, None)
 else:
 # single dtype or None
 cast_type = dtypes

 if self.na_filter:
 col_na_values, col_na_fvalues = _get_na_values(
 c, na_values, na_fvalues, self.keep_default_na
)
 else:
 col_na_values, col_na_fvalues = set(), set()

 if conv_f is not None:
 # conv_f applied to data before inference
 if cast_type is not None:
 warnings.warn(
 (
 "Both a converter and dtype were specified "
 f"for column {c} - only the converter will be used"
),
 ParserWarning,
 stacklevel=7,
)

 try:
 values = lib.map_infer(values, conv_f)
 except ValueError:
 mask = algorithms.isin(values, list(na_values)).view(np.uint8)
 values = lib.map_infer_mask(values, conv_f, mask)

 cvals, na_count = self._infer_types(
 values, set(col_na_values) | col_na_fvalues, try_num_bool=False
)
 else:
 is_str_or_ea_dtype = is_string_dtype(
 cast_type
) or is_extension_array_dtype(cast_type)
 # skip inference if specified dtype is object
 # or casting to an EA
 try_num_bool = not (cast_type and is_str_or_ea_dtype)

 # general type inference and conversion
 cvals, na_count = self._infer_types(
 values, set(col_na_values) | col_na_fvalues, try_num_bool
)

 # type specified in dtype param or cast_type is an EA

```

```

 if cast_type and (
 not is_dtype_equal(cvals, cast_type)
 or is_extension_array_dtype(cast_type)
):
 try:
 if (
 is_bool_dtype(cast_type)
 and not is_categorical_dtype(cast_type)
 and na_count > 0
):
 raise ValueError(f"Bool column has NA values in column {c}")
 except (AttributeError, TypeError):
 # invalid input to is_bool_dtype
 pass
 cvals = self._cast_types(cvals, cast_type, c)

 result[c] = cvals
 if verbose and na_count:
 print(f"Filled {na_count} NA values in column {c!s}")
 return result

def _infer_types(self, values, na_values, try_num_bool=True):
 """
 Infer types of values, possibly casting

 Parameters

 values : ndarray
 na_values : set
 try_num_bool : bool, default try
 try to cast values to numeric (first preference) or boolean

 Returns

 converted : ndarray
 na_count : int
 """
 na_count = 0
 if issubclass(values.dtype.type, (np.number, np.bool_)):
 mask = algorithms.isin(values, list(na_values))
 na_count = mask.sum()
 if na_count > 0:
 if is_integer_dtype(values):
 values = values.astype(np.float64)
 np.putmask(values, mask, np.nan)
 return values, na_count

 if try_num_bool and is_object_dtype(values.dtype):
 # exclude e.g DatetimeIndex here
 try:
 result = lib.maybe_convert_numeric(values, na_values, False)
 except (ValueError, TypeError):
 # e.g. encountering datetime string gets ValueError
 # TypeError can be raised in floatify
 result = values
 na_count = parsers.sanitize_objects(result, na_values, False)
 else:
 na_count = isna(result).sum()
 else:
 result = values
 if values.dtype == np.object_:
 na_count = parsers.sanitize_objects(values, na_values, False)

```

```

if result.dtype == np.object_ and try_num_bool:
 result = libops.maybe_convert_bool(
 np.asarray(values),
 true_values=self.true_values,
 false_values=self.false_values,
)

return result, na_count

def _cast_types(self, values, cast_type, column):
 """
 Cast values to specified type

 Parameters

 values : ndarray
 cast_type : string or np.dtype
 dtype to cast values to
 column : string
 column name - used only for error reporting

 Returns

 converted : ndarray
 """
 if is_categorical_dtype(cast_type):
 known_cats = (
 isinstance(cast_type, CategoricalDtype)
 and cast_type.categories is not None
)

 if not is_object_dtype(values) and not known_cats:
 # XXX this is for consistency with
 # c-parser which parses all categories
 # as strings
 values = astype_nansafe(values, str)

 cats = Index(values).unique().dropna()
 values = Categorical._from_inferred_categories(
 cats, cats.get_indexer(values), cast_type, true_values=self.true_values
)

 # use the EA's implementation of casting
 elif is_extension_array_dtype(cast_type):
 # ensure cast_type is an actual dtype and not a string
 cast_type = pandas_dtype(cast_type)
 array_type = cast_type.construct_array_type()
 try:
 return array_type._from_sequence_of_strings(values, dtype=cast_type)
 except NotImplementedError as err:
 raise NotImplementedError(
 f"Extension Array: {array_type} must implement "
 f"_from_sequence_of_strings in order to be used in parser methods"
) from err

 else:
 try:
 values = astype_nansafe(values, cast_type, copy=True, skipna=True)
 except ValueError as err:
 raise ValueError(
 f"Unable to convert column {column} to type {cast_type}"
) from err
return values

```

```

def _do_date_conversions(self, names, data):
 # returns data, columns

 if self.parse_dates is not None:
 data, names = _process_date_conversion(
 data,
 self._date_conv,
 self.parse_dates,
 self.index_col,
 self.index_names,
 names,
 keep_date_col=self.keep_date_col,
)

 return names, data

class CParserWrapper(ParserBase):
 """
 """

 def __init__(self, src, **kwds):
 self.kwds = kwds
 kwds = kwds.copy()

 ParserBase.__init__(self, kwds)

 encoding = kwds.get("encoding")

 if kwds.get("compression") is None and encoding:
 if isinstance(src, str):
 src = open(src, "rb")
 self.handles.append(src)

 # Handle the file object with universal line mode enabled.
 # We will handle the newline character ourselves later on.
 if hasattr(src, "read") and not hasattr(src, "encoding"):
 src = TextIOWrapper(src, encoding=encoding, newline="")

 kwds["encoding"] = "utf-8"

 # #2442
 kwds["allow_leading_cols"] = self.index_col is not False

 # GH20529, validate usecol arg before TextReader
 self.usecols, self.usecols_dtype = _validate_usecols_arg(kwds["usecols"])
 kwds["usecols"] = self.usecols

 self._reader = parsers.TextReader(src, **kwds)
 self.unnamed_cols = self._reader.unnamed_cols

 passed_names = self.names is None

 if self._reader.header is None:
 self.names = None
 else:
 if len(self._reader.header) > 1:
 # we have a multi index in the columns
 (
 self.names,
 self.index_names,

```

```

 self.col_names,
 passed_names,
) = self._extract_multi_indexer_columns(
 self._reader.header, self.index_names, self.col_names, passed_names
)
else:
 self.names = list(self._reader.header[0])

if self.names is None:
 if self.prefix:
 self.names = [
 f"{self.prefix}{i}" for i in range(self._reader.table_width)
]
 else:
 self.names = list(range(self._reader.table_width))

gh-9755
#
need to set orig_names here first
so that proper indexing can be done
with _set_noconvert_columns
#
once names has been filtered, we will
then set orig_names again to names
self.orig_names = self.names[:]

if self.usecols:
 usecols = _evaluate_usecols(self.usecols, self.orig_names)

 # GH 14671
 if self.usecols_dtype == "string" and not set(usecols).issubset(
 self.orig_names
):
 _validate_usecols_names(usecols, self.orig_names)

 if len(self.names) > len(usecols):
 self.names = [
 n
 for i, n in enumerate(self.names)
 if (i in usecols or n in usecols)
]

 if len(self.names) < len(usecols):
 _validate_usecols_names(usecols, self.names)

self._validate_parse_dates_presence(self.names)
self._set_noconvert_columns()

self.orig_names = self.names

if not self._has_complex_date_col:
 if self._reader.leading_cols == 0 and _is_index_col(self.index_col):

 self._name_processed = True
 (index_names, self.names, self.index_col) = _clean_index_names(
 self.names, self.index_col, self.unnamed_cols
)

 if self.index_names is None:
 self.index_names = index_names

 if self._reader.header is None and not passed_names:
 self.index_names = [None] * len(self.index_names)

```

```

 self._implicit_index = self._reader.leading_cols > 0

def close(self):
 for f in self.handles:
 f.close()

 # close additional handles opened by C parser (for compression)
try:
 self._reader.close()
except ValueError:
 pass

def _set_noconvert_columns(self):
 """
 Set the columns that should not undergo dtype conversions.

 Currently, any column that is involved with date parsing will not
 undergo such conversions.
 """
 names = self.orig_names
 if self.usecols_dtype == "integer":
 # A set of integers will be converted to a list in
 # the correct order every single time.
 usecols = list(self.usecols)
 usecols.sort()
 elif callable(self.usecols) or self.usecols_dtype not in ("empty", None):
 # The names attribute should have the correct columns
 # in the proper order for indexing with parse_dates.
 usecols = self.names[:]
 else:
 # Usecols is empty.
 usecols = None

 def _set(x):
 if usecols is not None and is_integer(x):
 x = usecols[x]

 if not is_integer(x):
 x = names.index(x)

 self._reader.set_noconvert(x)

 if isinstance(self.parse_dates, list):
 for val in self.parse_dates:
 if isinstance(val, list):
 for k in val:
 _set(k)
 else:
 _set(val)

 elif isinstance(self.parse_dates, dict):
 for val in self.parse_dates.values():
 if isinstance(val, list):
 for k in val:
 _set(k)
 else:
 _set(val)

 elif self.parse_dates:
 if isinstance(self.index_col, list):
 for k in self.index_col:
 _set(k)

```

```

 elif self.index_col is not None:
 _set(self.index_col)

def set_error_bad_lines(self, status):
 self._reader.set_error_bad_lines(int(status))

def read(self, nrows=None):
 try:
 data = self._reader.read(nrows)
 except StopIteration:
 if self._first_chunk:
 self._first_chunk = False
 names = self._maybe_dedup_names(self.orig_names)
 index, columns, col_dict = _get_empty_meta(
 names,
 self.index_col,
 self.index_names,
 dtype=self.kwds.get("dtype"),
)
 columns = self._maybe_make_multi_index_columns(columns, self.col_names)

 if self.usecols is not None:
 columns = self._filter_usecols(columns)

 col_dict = dict(
 filter(lambda item: item[0] in columns, col_dict.items())
)

 return index, columns, col_dict

 else:
 raise

Done with first read, next time raise StopIteration
self._first_chunk = False

names = self.names

if self._reader.leading_cols:
 if self._has_complex_date_col:
 raise NotImplementedError("file structure not yet supported")

 # implicit index, no index names
 arrays = []

 for i in range(self._reader.leading_cols):
 if self.index_col is None:
 values = data.pop(i)
 else:
 values = data.pop(self.index_col[i])

 values = self._maybe_parse_dates(values, i, try_parse_dates=True)
 arrays.append(values)

 index = ensure_index_from_sequences(arrays)

 if self.usecols is not None:
 names = self._filter_usecols(names)

 names = self._maybe_dedup_names(names)

 # rename dict keys
 data = sorted(data.items())

```

```

 data = {k: v for k, (i, v) in zip(names, data)}

 names, data = self._do_date_conversions(names, data)

 else:
 # rename dict keys
 data = sorted(data.items())

 # ugh, mutation
 names = list(self.orig_names)
 names = self._maybe_dedup_names(names)

 if self.usecols is not None:
 names = self._filter_usecols(names)

 # columns as list
 alldata = [x[1] for x in data]

 data = {k: v for k, (i, v) in zip(names, data)}

 names, data = self._do_date_conversions(names, data)
 index, names = self._make_index(data, alldata, names)

 # maybe create a mi on the columns
 names = self._maybe_make_multi_index_columns(names, self.col_names)

 return index, names, data

def _filter_usecols(self, names):
 # hackish
 usecols = _evaluate_usecols(self.usecols, names)
 if usecols is not None and len(names) != len(usecols):
 names = [
 name for i, name in enumerate(names) if i in usecols or name in usecols
]
 return names

def _get_index_names(self):
 names = list(self._reader.header[0])
 idx_names = None

 if self._reader.leading_cols == 0 and self.index_col is not None:
 (idx_names, names, self.index_col) = _clean_index_names(
 names, self.index_col, self.unnamed_cols
)

 return names, idx_names

def _maybe_parse_dates(self, values, index, try_parse_dates=True):
 if try_parse_dates and self._should_parse_dates(index):
 values = self._date_conv(values)
 return values

def TextParser(*args, **kwds):
 """
 Converts lists of lists/tuples into DataFrames with proper type inference
 and optional (e.g. string to datetime) conversion. Also enables iterating
 lazily over chunks of large files

 Parameters

 data : file-like object or list
 """

```

```

delimiter : separator character to use
dialect : str or csv.Dialect instance, optional
 Ignored if delimiter is longer than 1 character
names : sequence, default
header : int, default 0
 Row to use to parse column labels. Defaults to the first row. Prior
 rows will be discarded
index_col : int or list, optional
 Column or columns to use as the (possibly hierarchical) index
has_index_names: bool, default False
 True if the cols defined in index_col have an index name and are
 not in the header.
na_values : scalar, str, list-like, or dict, optional
 Additional strings to recognize as NA/NaN.
keep_default_na : bool, default True
thousands : str, optional
 Thousands separator
comment : str, optional
 Comment out remainder of line
parse_dates : bool, default False
keep_date_col : bool, default False
date_parser : function, optional
skiprows : list of integers
 Row numbers to skip
skipfooter : int
 Number of line at bottom of file to skip
converters : dict, optional
 Dict of functions for converting values in certain columns. Keys can
 either be integers or column labels, values are functions that take one
 input argument, the cell (not column) content, and return the
 transformed content.
encoding : str, optional
 Encoding to use for UTF when reading/writing (ex. 'utf-8')
squeeze : bool, default False
 returns Series if only one column.
infer_datetime_format: bool, default False
 If True and `parse_dates` is True for a column, try to infer the
 datetime format based on the first datetime string. If the format
 can be inferred, there often will be a large parsing speed-up.
float_precision : str, optional
 Specifies which converter the C engine should use for floating-point
 values. The options are None for the ordinary converter,
 'high' for the high-precision converter, and 'round_trip' for the
 round-trip converter.

"""
kwds["engine"] = "python"
return TextFileReader(*args, **kwds)

def count_empty_vals(vals):
 return sum(1 for v in vals if v == "" or v is None)

class PythonParser(ParserBase):
 def __init__(self, f, **kwds):
 """
 Workhorse function for processing nested list into DataFrame
 """
 ParserBase.__init__(self, kwds)

 self.data = None
 self.buf = []
 self.pos = 0

```

```

self.line_pos = 0

self.encoding = kwds["encoding"]
self.compression = kwds["compression"]
self.memory_map = kwds["memory_map"]
self.skiprows = kwds["skiprows"]

if callable(self.skiprows):
 self.skipfunc = self.skiprows
else:
 self.skipfunc = lambda x: x in self.skiprows

self.skipfooter = _validate_skipfooter_arg(kwds["skipfooter"])
self.delimiter = kwds["delimiter"]

self.quotechar = kwds["quotechar"]
if isinstance(self.quotechar, str):
 self.quotechar = str(self.quotechar)

self.escapechar = kwds["escapechar"]
self.doublequote = kwds["doublequote"]
self.skipinitialspace = kwds["skipinitialspace"]
self.lineterminator = kwds["lineterminator"]
self.quoting = kwds["quoting"]
self.usecols, _ = _validate_usecols_arg(kwds["usecols"])
self.skip_blank_lines = kwds["skip_blank_lines"]

self.warn_bad_lines = kwds["warn_bad_lines"]
self.error_bad_lines = kwds["error_bad_lines"]

self.names_passed = kwds["names"] or None

self.has_index_names = False
if "has_index_names" in kwds:
 self.has_index_names = kwds["has_index_names"]

self.verbose = kwds["verbose"]
self.converters = kwds["converters"]

self.dtype = kwds["dtype"]
self.thousands = kwds["thousands"]
self.decimal = kwds["decimal"]

self.comment = kwds["comment"]
self._comment_lines = []

f, handles = get_handle(
 f,
 "r",
 encoding=self.encoding,
 compression=self.compression,
 memory_map=self.memory_map,
)
self.handles.extend(handles)

Set self.data to something that can read lines.
if hasattr(f, "readline"):
 self._make_reader(f)
else:
 self.data = f

Get columns in two steps: infer from data, then
infer column indices from self.usecols if it is specified.

```

```

self._col_indices = None
try:
 (
 self.columns,
 self.num_original_columns,
 self.unnamed_cols,
) = self._infer_columns()
except (TypeError, ValueError):
 self.close()
 raise

Now self.columns has the set of columns that we will process.
The original set is stored in self.original_columns.
if len(self.columns) > 1:
 # we are processing a multi index column
 (
 self.columns,
 self.index_names,
 self.col_names,
) = self._extract_multi_indexer_columns(
 self.columns, self.index_names, self.col_names
)
 # Update list of original names to include all indices.
 self.num_original_columns = len(self.columns)
else:
 self.columns = self.columns[0]

get popped off for index
self.orig_names = list(self.columns)

needs to be cleaned/refactored
multiple date column thing turning into a real spaghetti factory

if not self._has_complex_date_col:
 (index_names, self.orig_names, self.columns) = self._get_index_name(
 self.columns
)
 self._name_processed = True
 if self.index_names is None:
 self.index_names = index_names

 self._validate_parse_dates_presence(self.columns)
if self.parse_dates:
 self._no_thousands_columns = self._set_no_thousands_columns()
else:
 self._no_thousands_columns = None

if len(self.decimal) != 1:
 raise ValueError("Only length-1 decimal markers supported")

if self.thousands is None:
 self.nonnum = re.compile(fr"[^-^0-9^{{self.decimal}}]+")
else:
 self.nonnum = re.compile(fr"[^-^0-9^{{self.thousands}}^{{self.decimal}}]+")

def _set_no_thousands_columns(self):
 # Create a set of column ids that are not to be stripped of thousands
 # operators.
 noconvert_columns = set()

 def _set(x):
 if is_integer(x):

```

```

 noconvert_columns.add(x)
 else:
 noconvert_columns.add(self.columns.index(x))

 if isinstance(self.parse_dates, list):
 for val in self.parse_dates:
 if isinstance(val, list):
 for k in val:
 _set(k)
 else:
 _set(val)

 elif isinstance(self.parse_dates, dict):
 for val in self.parse_dates.values():
 if isinstance(val, list):
 for k in val:
 _set(k)
 else:
 _set(val)

 elif self.parse_dates:
 if isinstance(self.index_col, list):
 for k in self.index_col:
 _set(k)
 elif self.index_col is not None:
 _set(self.index_col)

 return noconvert_columns

def _make_reader(self, f):
 sep = self.delimiter

 if sep is None or len(sep) == 1:
 if self.lineterminator:
 raise ValueError(
 "Custom line terminators not supported in python parser (yet)"
)

 class MyDialect(csv.Dialect):
 delimiter = self.delimiter
 quotechar = self.quotechar
 escapechar = self.escapechar
 doublequote = self.doublequote
 skipinitialspace = self.skipinitialspace
 quoting = self.quoting
 lineterminator = "\n"

 dia = MyDialect

 if sep is not None:
 dia.delimiter = sep
 else:
 # attempt to sniff the delimiter from the first valid line,
 # i.e. no comment line and not in skiprows
 line = f.readline()
 lines = self._check_comments([[line]])[0]
 while self.skipfunc(self.pos) or not lines:
 self.pos += 1
 line = f.readline()
 lines = self._check_comments([[line]])[0]

 # since `line` was a string, lines will be a list containing
 # only a single string

```

```

line = lines[0]

 self.pos += 1
 self.line_pos += 1
 sniffed = csv.Sniffer().sniff(line)
 dia.delimiter = sniffed.delimiter

 # Note: self.encoding is irrelevant here
 line_rdr = csv.reader(StringIO(line), dialect=dia)
 self.buf.extend(list(line_rdr))

 # Note: self.encoding is irrelevant here
 reader = csv.reader(f, dialect=dia, strict=True)

else:

 def _read():
 line = f.readline()
 pat = re.compile(sep)

 yield pat.split(line.strip())

 for line in f:
 yield pat.split(line.strip())

 reader = _read()

self.data = reader

def read(self, rows=None):
 try:
 content = self._get_lines(rows)
 except StopIteration:
 if self._first_chunk:
 content = []
 else:
 raise

 # done with first read, next time raise StopIteration
 self._first_chunk = False

 columns = list(self.orig_names)
 if not len(content): # pragma: no cover
 # DataFrame with the right metadata, even though it's length 0
 names = self._maybe_dedup_names(self.orig_names)
 index, columns, col_dict = _get_empty_meta(
 names, self.index_col, self.index_names, self.dtype
)
 columns = self._maybe_make_multi_index_columns(columns, self.col_names)
 return index, columns, col_dict

 # handle new style for names in index
 count_empty_content_vals = count_empty_vals(content[0])
 indexnamerow = None
 if self.has_index_names and count_empty_content_vals == len(columns):
 indexnamerow = content[0]
 content = content[1:]

 alldata = self._rows_to_cols(content)
 data = self._exclude_implicit_index(alldata)

 columns = self._maybe_dedup_names(self.columns)
 columns, data = self._do_date_conversions(columns, data)

```

```

 data = self._convert_data(data)
 index, columns = self._make_index(data, alldata, columns, indexnamerow)

 return index, columns, data

def _exclude_implicit_index(self, alldata):
 names = self._maybe_dedup_names(self.orig_names)

 if self._implicit_index:
 excl_indices = self.index_col

 data = {}
 offset = 0
 for i, col in enumerate(names):
 while i + offset in excl_indices:
 offset += 1
 data[col] = alldata[i + offset]
 else:
 data = {k: v for k, v in zip(names, alldata)}

 return data

legacy
def get_chunk(self, size=None):
 if size is None:
 size = self.chunkszie
 return self.read(rows=size)

def _convert_data(self, data):
 # apply converters
 def _clean_mapping(mapping):
 """converts col numbers to names"""
 clean = {}
 for col, v in mapping.items():
 if isinstance(col, int) and col not in self.orig_names:
 col = self.orig_names[col]
 clean[col] = v
 return clean

 clean_conv = _clean_mapping(self.converters)
 if not isinstance(self.dtype, dict):
 # handles single dtype applied to all columns
 clean_dtypes = self.dtype
 else:
 clean_dtypes = _clean_mapping(self.dtype)

 # Apply NA values.
 clean_na_values = {}
 clean_na_fvalues = {}

 if isinstance(self.na_values, dict):
 for col in self.na_values:
 na_value = self.na_values[col]
 na_fvalue = self.na_fvalues[col]

 if isinstance(col, int) and col not in self.orig_names:
 col = self.orig_names[col]

 clean_na_values[col] = na_value
 clean_na_fvalues[col] = na_fvalue
 else:
 clean_na_values = self.na_values

```

```

 clean_na_fvalues = self.na_fvalues

 return self._convert_to_ndarrays(
 data,
 clean_na_values,
 clean_na_fvalues,
 self.verbose,
 clean_conv,
 clean_dtypes,
)

def _infer_columns(self):
 names = self.names
 num_original_columns = 0
 clear_buffer = True
 unnamed_cols = set()

 if self.header is not None:
 header = self.header

 if isinstance(header, (list, tuple, np.ndarray)):
 have_mi_columns = len(header) > 1
 # we have a mi columns, so read an extra line
 if have_mi_columns:
 header = list(header) + [header[-1] + 1]
 else:
 have_mi_columns = False
 header = [header]

 columns = []
 for level, hr in enumerate(header):
 try:
 line = self._buffered_line()

 while self.line_pos <= hr:
 line = self._next_line()

 except StopIteration as err:
 if self.line_pos < hr:
 raise ValueError(
 f"Passed header={hr} but only {self.line_pos + 1} lines in "
 "'file'"
) from err

 # We have an empty file, so check
 # if columns are provided. That will
 # serve as the 'line' for parsing
 if have_mi_columns and hr > 0:
 if clear_buffer:
 self._clear_buffer()
 columns.append([None] * len(columns[-1]))
 return columns, num_original_columns, unnamed_cols

 if not self.names:
 raise EmptyDataError("No columns to parse from file") from err

 line = self.names[:]

 this_columns = []
 this_unnamed_cols = []

 for i, c in enumerate(line):
 if c == "":

```

```

 if have_mi_columns:
 col_name = f"Unnamed: {i}_level_{level}"
 else:
 col_name = f"Unnamed: {i}"

 this_unnamed_cols.append(i)
 this_columns.append(col_name)
 else:
 this_columns.append(c)

if not have_mi_columns and self.mangle_dupe_cols:
 counts = defaultdict(int)

 for i, col in enumerate(this_columns):
 cur_count = counts[col]

 while cur_count > 0:
 counts[col] = cur_count + 1
 col = f"{col}.{cur_count}"
 cur_count = counts[col]

 this_columns[i] = col
 counts[col] = cur_count + 1
elif have_mi_columns:

 # if we have grabbed an extra line, but its not in our
 # format so save in the buffer, and create an blank extra
 # line for the rest of the parsing code
 if hr == header[-1]:
 lc = len(this_columns)
 ic = len(self.index_col) if self.index_col is not None else 0
 unnamed_count = len(this_unnamed_cols)

 if lc != unnamed_count and lc - ic > unnamed_count:
 clear_buffer = False
 this_columns = [None] * lc
 self.buf = [self.buf[-1]]

columns.append(this_columns)
unnamed_cols.update({this_columns[i] for i in this_unnamed_cols})

if len(columns) == 1:
 num_original_columns = len(this_columns)

if clear_buffer:
 self._clear_buffer()

if names is not None:
 if (self.usecols is not None and len(names) != len(self.usecols)) or (
 self.usecols is None and len(names) != len(columns[0]))
):
 raise ValueError(
 "Number of passed names did not match "
 "number of header fields in the file"
)
 if len(columns) > 1:
 raise TypeError("Cannot pass names with multi-index columns")

if self.usecols is not None:
 # Set _use_cols. We don't store columns because they are
 # overwritten.
 self._handle_usecols(columns, names)
else:

```

```

 self._col_indices = None
 num_original_columns = len(names)
 columns = [names]
 else:
 columns = self._handle_usecols(columns, columns[0])
else:
 try:
 line = self._buffered_line()

 except StopIteration as err:
 if not names:
 raise EmptyDataError("No columns to parse from file") from err

 line = names[:]

 ncols = len(line)
 num_original_columns = ncols

 if not names:
 if self.prefix:
 columns = [[f"{self.prefix}{i}" for i in range(ncols)]]
 else:
 columns = [list(range(ncols))]
 columns = self._handle_usecols(columns, columns[0])
 else:
 if self.usecols is None or len(names) >= num_original_columns:
 columns = self._handle_usecols([names], names)
 num_original_columns = len(names)
 else:
 if not callable(self.usecols) and len(names) != len(self.usecols):
 raise ValueError(
 "Number of passed names did not match number of "
 "header fields in the file"
)
 # Ignore output but set used columns.
 self._handle_usecols([names], names)
 columns = [names]
 num_original_columns = ncols

 return columns, num_original_columns, unnamed_cols

def _handle_usecols(self, columns, usecols_key):
 """
 Sets self._col_indices

 usecols_key is used if there are string usecols.
 """
 if self.usecols is not None:
 if callable(self.usecols):
 col_indices = _evaluate_usecols(self.usecols, usecols_key)
 elif any(isinstance(u, str) for u in self.usecols):
 if len(columns) > 1:
 raise ValueError(
 "If using multiple headers, usecols must be integers."
)
 col_indices = []
 for col in self.usecols:
 if isinstance(col, str):
 try:
 col_indices.append(usecols_key.index(col))
 except ValueError:
 validate_usecols_names(self.usecols, usecols_key)
 else:
 col_indices = None
 self._col_indices = col_indices

```

```

 else:
 col_indices.append(col)
 else:
 col_indices = self.usecols

 columns = [
 [n for i, n in enumerate(column) if i in col_indices]
 for column in columns
]
 self._col_indices = col_indices
return columns

def _buffered_line(self):
 """
 Return a line from buffer, filling buffer if required.
 """
 if len(self.buf) > 0:
 return self.buf[0]
 else:
 return self._next_line()

def _check_for_bom(self, first_row):
 """
 Checks whether the file begins with the BOM character.
 If it does, remove it. In addition, if there is quoting
 in the field subsequent to the BOM, remove it as well
 because it technically takes place at the beginning of
 the name, not the middle of it.
 """
 # first_row will be a list, so we need to check
 # that that list is not empty before proceeding.
 if not first_row:
 return first_row

 # The first element of this row is the one that could have the
 # BOM that we want to remove. Check that the first element is a
 # string before proceeding.
 if not isinstance(first_row[0], str):
 return first_row

 # Check that the string is not empty, as that would
 # obviously not have a BOM at the start of it.
 if not first_row[0]:
 return first_row

 # Since the string is non-empty, check that it does
 # in fact begin with a BOM.
 first_elt = first_row[0][0]
 if first_elt != _BOM:
 return first_row

 first_row_bom = first_row[0]

 if len(first_row_bom) > 1 and first_row_bom[1] == self.quotechar:
 start = 2
 quote = first_row_bom[1]
 end = first_row_bom[2:].index(quote) + 2

 # Extract the data between the quotation marks
 new_row = first_row_bom[start:end]

 # Extract any remaining data after the second
 # quotation mark.

```

```

 if len(first_row_bom) > end + 1:
 new_row += first_row_bom[end + 1 :]
 return [new_row] + first_row[1:]

 elif len(first_row_bom) > 1:
 return [first_row_bom[1:]]
 else:
 # First row is just the BOM, so we
 # return an empty string.
 return [""]

def _is_line_empty(self, line):
 """
 Check if a line is empty or not.

 Parameters

 line : str, array-like
 The line of data to check.

 Returns

 boolean : Whether or not the line is empty.
 """
 return not line or all(not x for x in line)

def _next_line(self):
 if isinstance(self.data, list):
 while self.skipfunc(self.pos):
 self.pos += 1

 while True:
 try:
 line = self._check_comments([self.data[self.pos]])[0]
 self.pos += 1
 # either uncommented or blank to begin with
 if not self.skip_blank_lines and (
 self._is_line_empty(self.data[self.pos - 1]) or line
):
 break
 elif self.skip_blank_lines:
 ret = self._remove_empty_lines([line])
 if ret:
 line = ret[0]
 break
 except IndexError:
 raise StopIteration
 else:
 while self.skipfunc(self.pos):
 self.pos += 1
 next(self.data)

 while True:
 orig_line = self._next_iter_line(row_num=self.pos + 1)
 self.pos += 1

 if orig_line is not None:
 line = self._check_comments([orig_line])[0]

 if self.skip_blank_lines:
 ret = self._remove_empty_lines([line])

 if ret:

```

```

 line = ret[0]
 break
 elif self._is_line_empty(orig_line) or line:
 break

 # This was the first line of the file,
 # which could contain the BOM at the
 # beginning of it.
 if self.pos == 1:
 line = self._check_for_bom(line)

 self.line_pos += 1
 self.buf.append(line)
 return line

def _alert_malformed(self, msg, row_num):
 """
 Alert a user about a malformed row.

 If `self.error_bad_lines` is True, the alert will be `ParserError`.
 If `self.warn_bad_lines` is True, the alert will be printed out.

 Parameters

 msg : The error message to display.
 row_num : The row number where the parsing error occurred.
 Because this row number is displayed, we 1-index,
 even though we 0-index internally.
 """
 if self.error_bad_lines:
 raise ParserError(msg)
 elif self.warn_bad_lines:
 base = f"Skipping line {row_num}: "
 sys.stderr.write(base + msg + "\n")

def _next_iter_line(self, row_num):
 """
 Wrapper around iterating through `self.data` (CSV source).

 When a CSV error is raised, we check for specific
 error messages that allow us to customize the
 error message displayed to the user.

 Parameters

 row_num : The row number of the line being parsed.
 """
 try:
 return next(self.data)
 except csv.Error as e:
 if self.warn_bad_lines or self.error_bad_lines:
 msg = str(e)

 if "NULL byte" in msg or "line contains NUL" in msg:
 msg = (
 "NULL byte detected. This byte "
 "cannot be processed in Python's "
 "native csv library at the moment, "
 "so please pass in engine='c' instead"
)

 if self.skipfooter > 0:
 reason = (

```

```

 "Error could possibly be due to "
 "parsing errors in the skipped footer rows "
 "(the skipfooter keyword is only applied "
 "after Python's csv library has parsed "
 "all rows)."
)
 msg += ". " + reason

 self._alert_malformed(msg, row_num)
 return None

def _check_comments(self, lines):
 if self.comment is None:
 return lines
 ret = []
 for l in lines:
 rl = []
 for x in l:
 if not isinstance(x, str) or self.comment not in x:
 rl.append(x)
 else:
 x = x[:x.find(self.comment)]
 if len(x) > 0:
 rl.append(x)
 break
 ret.append(rl)
 return ret

def _remove_empty_lines(self, lines):
 """
 Iterate through the lines and remove any that are
 either empty or contain only one whitespace value

 Parameters

 lines : array-like
 The array of lines that we are to filter.

 Returns

 filtered_lines : array-like
 The same array of lines with the "empty" ones removed.
 """
 ret = []
 for l in lines:
 # Remove empty lines and lines with only one whitespace value
 if (
 len(l) > 1
 or len(l) == 1
 and (not isinstance(l[0], str) or l[0].strip())
):
 ret.append(l)
 return ret

def _check_thousands(self, lines):
 if self.thousands is None:
 return lines

 return self._search_replace_num_columns(
 lines=lines, search=self.thousands, replace=""
)

def _search_replace_num_columns(self, lines, search, replace):

```

```

ret = []
for l in lines:
 rl = []
 for i, x in enumerate(l):
 if (
 not isinstance(x, str)
 or search not in x
 or (self._no_thousands_columns and i in self._no_thousands_columns)
 or self.nonnum.search(x.strip())
):
 rl.append(x)
 else:
 rl.append(x.replace(search, replace))
 ret.append(rl)
return ret

def _check_decimal(self, lines):
 if self.decimal == _parser_defaults["decimal"]:
 return lines

 return self._search_replace_num_columns(
 lines=lines, search=self.decimal, replace="."
)

def _clear_buffer(self):
 self.buf = []

_implicit_index = False

def _get_index_name(self, columns):
 """
 Try several cases to get lines:

 0) There are headers on row 0 and row 1 and their
 total summed lengths equals the length of the next line.
 Treat row 0 as columns and row 1 as indices
 1) Look for implicit index: there are more columns
 on row 1 than row 0. If this is true, assume that row
 1 lists index columns and row 0 lists normal columns.
 2) Get index from the columns if it was listed.
 """
 orig_names = list(columns)
 columns = list(columns)

 try:
 line = self._next_line()
 except StopIteration:
 line = None

 try:
 next_line = self._next_line()
 except StopIteration:
 next_line = None

 # implicitly index_col=0 b/c 1 fewer column names
 implicit_first_cols = 0
 if line is not None:
 # leave it 0, #2442
 # Case 1
 if self.index_col is not False:
 implicit_first_cols = len(line) - self.num_original_columns

 # Case 0

```

```

 if next_line is not None:
 if len(next_line) == len(line) + self.num_original_columns:
 # column and index names on diff rows
 self.index_col = list(range(len(line)))
 self.buf = self.buf[1:]

 for c in reversed(line):
 columns.insert(0, c)

 # Update list of original names to include all indices.
 orig_names = list(columns)
 self.num_original_columns = len(columns)
 return line, orig_names, columns

 if implicit_first_cols > 0:
 # Case 1
 self._implicit_index = True
 if self.index_col is None:
 self.index_col = list(range(implicit_first_cols))

 index_name = None

 else:
 # Case 2
 (index_name, columns_, self.index_col) = _clean_index_names(
 columns, self.index_col, self.unnamed_cols
)

 return index_name, orig_names, columns

def _rows_to_cols(self, content):
 col_len = self.num_original_columns

 if self._implicit_index:
 col_len += len(self.index_col)

 max_len = max(len(row) for row in content)

 # Check that there are no rows with too many
 # elements in their row (rows with too few
 # elements are padded with NaN).
 if max_len > col_len and self.index_col is not False and self.usecols is None:

 footers = self.skipfooter if self.skipfooter else 0
 bad_lines = []

 iter_content = enumerate(content)
 content_len = len(content)
 content = []

 for (i, l) in iter_content:
 actual_len = len(l)

 if actual_len > col_len:
 if self.error_bad_lines or self.warn_bad_lines:
 row_num = self.pos - (content_len - i + footers)
 bad_lines.append((row_num, actual_len))

 if self.error_bad_lines:
 break
 else:
 content.append(l)

```

```

 for row_num, actual_len in bad_lines:
 msg = (
 f"Expected {col_len} fields in line {row_num + 1}, saw "
 f"{actual_len}"
)
 if (
 self.delimiter
 and len(self.delimiter) > 1
 and self.quoting != csv.QUOTE_NONE
):
 # see gh-13374
 reason = (
 "Error could possibly be due to quotes being "
 "ignored when a multi-char delimiter is used."
)
 msg += ". " + reason

 self._alert_malformed(msg, row_num + 1)

 # see gh-13320
 zipped_content = list(lib.to_object_array(content, min_width=col_len).T)

 if self.usecols:
 if self._implicit_index:
 zipped_content = [
 a
 for i, a in enumerate(zipped_content)
 if (
 i < len(self.index_col)
 or i - len(self.index_col) in self._col_indices
)
]
 else:
 zipped_content = [
 a for i, a in enumerate(zipped_content) if i in self._col_indices
]
 return zipped_content

def _get_lines(self, rows=None):
 lines = self.buf
 new_rows = None

 # already fetched some number
 if rows is not None:
 # we already have the lines in the buffer
 if len(self.buf) >= rows:
 new_rows, self.buf = self.buf[:rows], self.buf[rows:]

 # need some lines
 else:
 rows -= len(self.buf)

 if new_rows is None:
 if isinstance(self.data, list):
 if self.pos > len(self.data):
 raise StopIteration
 if rows is None:
 new_rows = self.data[self.pos :]
 new_pos = len(self.data)
 else:
 new_rows = self.data[self.pos : self.pos + rows]
 new_pos = self.pos + rows

```

```

Check for stop rows. n.b.: self.skiprows is a set.
if self.skiprows:
 new_rows = [
 row
 for i, row in enumerate(new_rows)
 if not self.skipfunc(i + self.pos)
]

 lines.extend(new_rows)
 self.pos = new_pos

else:
 new_rows = []
 try:
 if rows is not None:
 for _ in range(rows):
 new_rows.append(next(self.data))
 lines.extend(new_rows)
 else:
 rows = 0

 while True:
 new_row = self._next_iter_line(row_num=self.pos + rows + 1)
 rows += 1

 if new_row is not None:
 new_rows.append(new_row)

 except StopIteration:
 if self.skiprows:
 new_rows = [
 row
 for i, row in enumerate(new_rows)
 if not self.skipfunc(i + self.pos)
]
 lines.extend(new_rows)
 if len(lines) == 0:
 raise
 self.pos += len(new_rows)

 self.buf = []
 else:
 lines = new_rows

if self.skipfooter:
 lines = lines[: -self.skipfooter]

lines = self._check_comments(lines)
if self.skip_blank_lines:
 lines = self._remove_empty_lines(lines)
lines = self._check_thousands(lines)
return self._check_decimal(lines)

def _make_date_converter(
 date_parser=None, dayfirst=False, infer_datetime_format=False, cache_dates=True
):
 def converter(*date_cols):
 if date_parser is None:
 strs = parsing._concat_date_cols(date_cols)

 try:
 return tools.to_datetime(

```

```

 ensure_object(strs),
 utc=None,
 dayfirst=dayfirst,
 errors="ignore",
 infer_datetime_format=infer_datetime_format,
 cache=cache_dates,
).to_numpy()

 except ValueError:
 return tools.to_datetime(
 parsing.try_parse_dates(strs, dayfirst=dayfirst), cache=cache_dates
)
 else:
 try:
 result = tools.to_datetime(
 date_parser(*date_cols), errors="ignore", cache=cache_dates
)
 if isinstance(result, datetime.datetime):
 raise Exception("scalar parser")
 return result
 except Exception:
 try:
 return tools.to_datetime(
 parsing.try_parse_dates(
 parsing._concat_date_cols(date_cols),
 parser=date_parser,
 dayfirst=dayfirst,
),
 errors="ignore",
)
 except Exception:
 return generic_parser(date_parser, *date_cols)

 return converter

def _process_date_conversion(
 data_dict,
 converter,
 parse_spec,
 index_col,
 index_names,
 columns,
 keep_date_col=False,
):
 def _isindex(colspec):
 return (isinstance(index_col, list) and colspec in index_col) or (
 isinstance(index_names, list) and colspec in index_names
)

 new_cols = []
 new_data = {}

 orig_names = columns
 columns = list(columns)

 date_cols = set()

 if parse_spec is None or isinstance(parse_spec, bool):
 return data_dict, columns

 if isinstance(parse_spec, list):
 # list of column lists

```

```

for colspec in parse_spec:
 if is_scalar(colspec):
 if isinstance(colspec, int) and colspec not in data_dict:
 colspec = orig_names[colspec]
 if _isindex(colspec):
 continue
 data_dict[colspec] = converter(data_dict[colspec])
else:
 new_name, col, old_names = _try_convert_dates(
 converter, colspec, data_dict, orig_names
)
 if new_name in data_dict:
 raise ValueError(f"New date column already in dict {new_name}")
 new_data[new_name] = col
 new_cols.append(new_name)
 date_cols.update(old_names)

elif isinstance(parse_spec, dict):
 # dict of new name to column list
 for new_name, colspec in parse_spec.items():
 if new_name in data_dict:
 raise ValueError(f"Date column {new_name} already in dict")

 _, col, old_names = _try_convert_dates(
 converter, colspec, data_dict, orig_names
)

 new_data[new_name] = col
 new_cols.append(new_name)
 date_cols.update(old_names)

data_dict.update(new_data)
new_cols.extend(columns)

if not keep_date_col:
 for c in list(date_cols):
 data_dict.pop(c)
 new_cols.remove(c)

return data_dict, new_cols

def _try_convert_dates(parser, colspec, data_dict, columns):
 colset = set(columns)
 colnames = []

 for c in colspec:
 if c in colset:
 colnames.append(c)
 elif isinstance(c, int) and c not in columns:
 colnames.append(columns[c])
 else:
 colnames.append(c)

 new_name = "_".join(str(x) for x in colnames)
 to_parse = [data_dict[c] for c in colnames if c in data_dict]

 new_col = parser(*to_parse)
 return new_name, new_col, colnames

def _clean_na_values(na_values, keep_default_na=True):

```

```

if na_values is None:
 if keep_default_na:
 na_values = STR_NA_VALUES
 else:
 na_values = set()
 na_fvalues = set()
elif isinstance(na_values, dict):
 old_na_values = na_values.copy()
 na_values = {} # Prevent aliasing.

 # Convert the values in the na_values dictionary
 # into array-likes for further use. This is also
 # where we append the default NaN values, provided
 # that `keep_default_na=True`.
 for k, v in old_na_values.items():
 if not is_list_like(v):
 v = [v]

 if keep_default_na:
 v = set(v) | STR_NA_VALUES

 na_values[k] = v
 na_fvalues = {k: _floatify_na_values(v) for k, v in na_values.items()}
else:
 if not is_list_like(na_values):
 na_values = [na_values]
 na_values = _stringify_na_values(na_values)
 if keep_default_na:
 na_values = na_values | STR_NA_VALUES

 na_fvalues = _floatify_na_values(na_values)

return na_values, na_fvalues

def _clean_index_names(columns, index_col, unnamed_cols):
 if not _is_index_col(index_col):
 return None, columns, index_col

 columns = list(columns)

 cp_cols = list(columns)
 index_names = []

 # don't mutate
 index_col = list(index_col)

 for i, c in enumerate(index_col):
 if isinstance(c, str):
 index_names.append(c)
 for j, name in enumerate(cp_cols):
 if name == c:
 index_col[i] = j
 columns.remove(name)
 break
 else:
 name = cp_cols[c]
 columns.remove(name)
 index_names.append(name)

 # Only clean index names that were placeholders.
 for i, name in enumerate(index_names):
 if isinstance(name, str) and name in unnamed_cols:

```

```

 index_names[i] = None

 return index_names, columns, index_col

def _get_empty_meta(columns, index_col, index_names, dtype=None):
 columns = list(columns)

 # Convert `dtype` to a defaultdict of some kind.
 # This will enable us to write `dtype[col_name]` `
 # without worrying about KeyError issues later on.
 if not isinstance(dtype, dict):
 # if dtype == None, default will be np.object.
 default_dtype = dtype or np.object_
 dtype = defaultdict(lambda: default_dtype)
 else:
 # Save a copy of the dictionary.
 _dtype = dtype.copy()
 dtype = defaultdict(lambda: np.object_)

 # Convert column indexes to column names.
 for k, v in _dtype.items():
 col = columns[k] if is_integer(k) else k
 dtype[col] = v

 # Even though we have no data, the "index" of the empty DataFrame
 # could for example still be an empty MultiIndex. Thus, we need to
 # check whether we have any index columns specified, via either:
 #
 # 1) index_col (column indices)
 # 2) index_names (column names)
 #
 # Both must be non-null to ensure a successful construction. Otherwise,
 # we have to create a generic empty Index.
 if (index_col is None or index_col is False) or index_names is None:
 index = Index([])
 else:
 data = [Series([], dtype=dtype[name]) for name in index_names]
 index = ensure_index_from_sequences(data, names=index_names)
 index_col.sort()

 for i, n in enumerate(index_col):
 columns.pop(n - i)

 col_dict = {col_name: Series([], dtype=dtype[col_name]) for col_name in columns}

 return index, columns, col_dict

def _floatify_na_values(na_values):
 # create float versions of the na_values
 result = set()
 for v in na_values:
 try:
 v = float(v)
 if not np.isnan(v):
 result.add(v)
 except (TypeError, ValueError, OverflowError):
 pass
 return result

def _stringify_na_values(na_values):

```

```

""" return a stringified and numeric for these values """
result = []
for x in na_values:
 result.append(str(x))
 result.append(x)
try:
 v = float(x)

 # we are like 999 here
 if v == int(v):
 v = int(v)
 result.append(f"{v}.0")
 result.append(str(v))

 result.append(v)
except (TypeError, ValueError, OverflowError):
 pass
try:
 result.append(int(x))
except (TypeError, ValueError, OverflowError):
 pass
return set(result)

def _get_na_values(col, na_values, na_fvalues, keep_default_na):
"""
Get the NaN values for a given column.

Parameters

col : str
 The name of the column.
na_values : array-like, dict
 The object listing the NaN values as strings.
na_fvalues : array-like, dict
 The object listing the NaN values as floats.
keep_default_na : bool
 If `na_values` is a dict, and the column is not mapped in the
 dictionary, whether to return the default NaN values or the empty set.

Returns

nan_tuple : A length-two tuple composed of

 1) na_values : the string NaN values for that column.
 2) na_fvalues : the float NaN values for that column.
"""
if isinstance(na_values, dict):
 if col in na_values:
 return na_values[col], na_fvalues[col]
 else:
 if keep_default_na:
 return STR_NA_VALUES, set()

 return set(), set()
else:
 return na_values, na_fvalues

def _get_col_names(colspec, columns):
 colset = set(columns)
 colnames = []
 for c in colspect:

```

```

if c in colset:
 colnames.append(c)
elif isinstance(c, int):
 colnames.append(columns[c])
return colnames

class FixedWidthReader(abc.Iterator):
 """
 A reader of fixed-width lines.
 """

 def __init__(self, f, colspecs, delimiter, comment, skiprows=None, infer_nrows=100):
 self.f = f
 self.buffer = None
 self.delimiter = "\r\n" + delimiter if delimiter else "\n\r\t"
 self.comment = comment
 if colspecs == "infer":
 self.colspecs = self.detect_colspecs(
 infer_nrows=infer_nrows, skiprows=skiprows
)
 else:
 self.colspecs = colspecs

 if not isinstance(self.colspecs, (tuple, list)):
 raise TypeError(
 "column specifications must be a list or tuple, "
 f"input was a {type(colspecs).__name__}"
)

 for colspec in self.colspecs:
 if not (
 isinstance(colspec, (tuple, list))
 and len(colspec) == 2
 and isinstance(colspec[0], (int, np.integer, type(None)))
 and isinstance(colspec[1], (int, np.integer, type(None)))
):
 raise TypeError(
 "Each column specification must be "
 "2 element tuple or list of integers"
)

 def get_rows(self, infer_nrows, skiprows=None):
 """
 Read rows from self.f, skipping as specified.

 We distinguish buffer_rows (the first <= infer_nrows
 lines) from the rows returned to detect_colspecs
 because it's simpler to leave the other locations
 with skiprows logic alone than to modify them to
 deal with the fact we skipped some rows here as
 well.

 Parameters

 infer_nrows : int
 Number of rows to read from self.f, not counting
 rows that are skipped.
 skiprows: set, optional
 Indices of rows to skip.

 Returns

 """

```

```

detect_rows : list of str
 A list containing the rows to read.

"""
if skiprows is None:
 skiprows = set()
buffer_rows = []
detect_rows = []
for i, row in enumerate(self.f):
 if i not in skiprows:
 detect_rows.append(row)
 buffer_rows.append(row)
 if len(detect_rows) >= infer_nrows:
 break
self.buffer = iter(buffer_rows)
return detect_rows

def detect_colspecs(self, infer_nrows=100, skiprows=None):
 # Regex escape the delimiters
 delimiters = "".join(fr"\{x}" for x in self.delimiter)
 pattern = re.compile(f"({[^{delimiters}]+})")
 rows = self.get_rows(infer_nrows, skiprows)
 if not rows:
 raise EmptyDataError("No rows from which to infer column width")
 max_len = max(map(len, rows))
 mask = np.zeros(max_len + 1, dtype=int)
 if self.comment is not None:
 rows = [row.partition(self.comment)[0] for row in rows]
 for row in rows:
 for m in pattern.finditer(row):
 mask[m.start() : m.end()] = 1
 shifted = np.roll(mask, 1)
 shifted[0] = 0
 edges = np.where((mask ^ shifted) == 1)[0]
 edge_pairs = list(zip(edges[::2], edges[1::2]))
 return edge_pairs

def __next__(self):
 if self.buffer is not None:
 try:
 line = next(self.buffer)
 except StopIteration:
 self.buffer = None
 line = next(self.f)
 else:
 line = next(self.f)
 # Note: 'colspecs' is a sequence of half-open intervals.
 return [line[fromm:to].strip(self.delimiter) for (fromm, to) in self.colspecs]

class FixedWidthFieldParser(PythonParser):
 """
 Specialization that Converts fixed-width fields into DataFrames.
 See PythonParser for details.
 """

 def __init__(self, f, **kwds):
 # Support iterators, convert to a list.
 self.colspecs = kwds.pop("colspecs")
 self.infer_nrows = kwds.pop("infer_nrows")
 PythonParser.__init__(self, f, **kwds)

 def make_reader(self, f):

```

```
self.data = FixedWidthReader(
 f,
 self.colspecs,
 self.delimiter,
 self.comment,
 self.skiprows,
 self.infer_nrows,
)
```

<https://github.com/pandas-dev/pandas/blob/master/pandas/core/generic.py>

```
import collections
from datetime import timedelta
import functools
import gc
import json
import operator
import pickle
import re
from textwrap import dedent
from typing import (
 TYPE_CHECKING,
 Any,
 Callable,
 Dict,
 FrozenSet,
 Hashable,
 List,
 Mapping,
 Optional,
 Sequence,
 Set,
 Tuple,
 Type,
 Union,
)
import warnings
import weakref

import numpy as np

from pandas._config import config

from pandas._libs import Timestamp, lib
from pandas._typing import (
 Axis,
 FilePathOrBuffer,
 FrameOrSeries,
 JSONSerializable,
 Label,
 Level,
 Renamer,
)
from pandas.compat import set_function_name
from pandas.compat._optional import import_optional_dependency
from pandas.compat.numpy import function as nv
from pandas.errors import AbstractMethodError
from pandas.util._decorators import (
 Appender,
 Substitution,
 doc,
 rewrite_axis_style_signature,
)
from pandas.util._validators import (
 validate_bool_kwarg,
 validate_fillna_kwargs,
 validate_percentile,
)
```

```
from pandas.core.dtypes.common import (
 ensure_int64,
 ensure_object,
 ensure_str,
 is_bool,
 is_bool_dtype,
 is_datetime64_any_dtype,
 is_datetime64tz_dtype,
 is_dict_like,
 is_extension_array_dtype,
 is_float,
 is_list_like,
 is_number,
 is_numeric_dtype,
 is_object_dtype,
 is_re_compilable,
 is_scalar,
 is_timedelta64_dtype,
 pandas_dtype,
)
from pandas.core.dtypes.generic import ABCDataFrame, ABCSeries
from pandas.core.dtypes.inference import is_hashable
from pandas.core.dtypes.missing import isna, notna

import pandas as pd
from pandas.core import missing, nanops
import pandas.core.algorithms as algos
from pandas.core.base import PandasObject, SelectionMixin
import pandas.core.common as com
from pandas.core.construction import create_series_with_explicit_dtype
from pandas.core.indexes.api import (
 Index,
 InvalidIndexError,
 MultiIndex,
 RangeIndex,
 ensure_index,
)
from pandas.core.indexes.datetimes import DatetimeIndex
from pandas.core.indexes.period import Period, PeriodIndex
import pandas.core.indexing as indexing
from pandas.core.internals import BlockManager
from pandas.core.missing import find_valid_index
from pandas.core.ops import _align_method_FRAME

from pandas.io.formats import format as fmt
from pandas.io.formats.format import DataFrameFormatter, format_percentiles
from pandas.io.formats.printing import pprint_thing
from pandas.tseries.frequencies import to_offset
from pandas.tseries.offsets import Tick

if TYPE_CHECKING:
 from pandas.core.resample import Resampler

goal is to be able to define the docs close to function, while still being
able to share
_shared_docs: Dict[str, str] = dict()
_shared_doc_kwargs = dict(
 axes="keywords for axes",
 klass="Series/DataFrame",
 axes_single_arg="int or labels for object",
 args_transpose="axes to permute (int or label for object)",
 optional_by="""

```

```

 by : str or list of str
 Name or list of names to sort by"",
)

def _single_replace(self, to_replace, method, inplace, limit):
 """
 Replaces values in a Series using the fill method specified when no
 replacement value is given in the replace method
 """
 if self.ndim != 1:
 raise TypeError(
 f"cannot replace {to_replace} with method {method} on a "
 f"{type(self).__name__}"
)

 orig_dtype = self.dtype
 result = self if inplace else self.copy()
 fill_f = missing.get_fill_func(method)

 mask = missing.mask_missing(result.values, to_replace)
 values = fill_f(result.values, limit=limit, mask=mask)

 if values.dtype == orig_dtype and inplace:
 return

 result = pd.Series(values, index=self.index, dtype=self.dtype).__finalize__(self)

 if inplace:
 self._update_inplace(result)
 return

 return result

bool_t = bool # Need alias because NDFrame has def bool:

class NDFrame(PandasObject, SelectionMixin, indexing.IndexingMixin):
 """
 N-dimensional analogue of DataFrame. Store multi-dimensional in a
 size-mutable, labeled data structure

 Parameters

 data : BlockManager
 axes : list
 copy : bool, default False
 """

 _internal_names: List[str] = [
 "_mgr",
 "_cacher",
 "_item_cache",
 "_cache",
 "_is_copy",
 "_subtyp",
 "_name",
 "_index",
 "_default_kind",
 "_default_fill_value",
 "_metadata",
 "_array_struct_",
]

```

```

 "__array_interface__",
]
 _internal_names_set: Set[str] = set(_internal_names)
 _accessors: Set[str] = set()
 _deprecations: FrozenSet[str] = frozenset(["get_values"])
 _metadata: List[str] = []
 _is_copy = None
 _mgr: BlockManager
 _attrs: Dict[Optional[Hashable], Any]
 _typ: str

Constructors

def __init__(
 self,
 data: BlockManager,
 copy: bool = False,
 attrs: Optional[Mapping[Optional[Hashable], Any]] = None,
):
 # copy kwarg is retained for mypy compat, is not used

 object.__setattr__(self, "_is_copy", None)
 object.__setattr__(self, "_mgr", data)
 object.__setattr__(self, "_item_cache", {})
 if attrs is None:
 attrs = {}
 else:
 attrs = dict(attrs)
 object.__setattr__(self, "_attrs", attrs)

@classmethod
def __init_mgr(cls, mgr, axes, dtype=None, copy: bool = False) -> BlockManager:
 """ passed a manager and a axes dict """
 for a, axe in axes.items():
 if axe is not None:
 axe = ensure_index(axe)
 bm_axis = cls._get_block_manager_axis(a)
 mgr = mgr.reindex_axis(axe, axis=bm_axis, copy=False)

 # make a copy if explicitly requested
 if copy:
 mgr = mgr.copy()
 if dtype is not None:
 # avoid further copies if we can
 if len(mgr.blocks) > 1 or mgr.blocks[0].values.dtype != dtype:
 mgr = mgr.astype(dtype=dtype)
 return mgr

@property
def attrs(self) -> Dict[Optional[Hashable], Any]:
 """
 Dictionary of global attributes on this object.

 .. warning::

 attrs is experimental and may change without warning.
 """
 if self._attrs is None:
 self._attrs = {}
 return self._attrs

```

```

@attrs.setter
def attrs(self, value: Mapping[Optional[Hashable], Any]) -> None:
 self._attrs = dict(value)

@classmethod
def _validate_dtype(cls, dtype):
 """ validate the passed dtype """
 if dtype is not None:
 dtype = pandas_dtype(dtype)

 # a compound dtype
 if dtype.kind == "V":
 raise NotImplementedError(
 "compound dtypes are not implemented "
 f"in the {cls.__name__} constructor"
)

 return dtype

Construction

@property
def _constructor(self: FrameOrSeries) -> Type[FrameOrSeries]:
 """
 Used when a manipulation result has the same dimensions as the
 original.
 """
 raise AbstractMethodError(self)

@property
def _constructor_sliced(self):
 """
 Used when a manipulation result has one lower dimension(s) as the
 original, such as DataFrame single columns slicing.
 """
 raise AbstractMethodError(self)

@property
def _constructor_expanddim(self):
 """
 Used when a manipulation result has one higher dimension as the
 original, such as Series.to_frame()
 """
 raise NotImplementedError

Internals

@property
def _data(self):
 # GH#33054 retained because some downstream packages uses this,
 # e.g. fastparquet
 return self._mgr

Axis
_stat_axis_number = 0
_stat_axis_name = "index"
_ix = None
_AXIS_ORDERS: List[str]
_AXIS_TO_AXIS_NUMBER: Dict[Axis, int] = {0: 0, "index": 0, "rows": 0}

```

```

_AXIS_REVERSED: bool
_info_axis_number: int
_info_axis_name: str
_AXIS_LEN: int

@property
def _AXIS_NUMBERS(self) -> Dict[str, int]:
 """.. deprecated:: 1.1.0"""
 warnings.warn(
 "_AXIS_NUMBERS has been deprecated.", FutureWarning, stacklevel=3,
)
 return {"index": 0}

@property
def _AXIS_NAMES(self) -> Dict[int, str]:
 """.. deprecated:: 1.1.0"""
 warnings.warn(
 "_AXIS_NAMES has been deprecated.", FutureWarning, stacklevel=3,
)
 return {0: "index"}

def _construct_axes_dict(self, axes=None, **kwargs):
 """Return an axes dictionary for myself."""
 d = {a: self._get_axis(a) for a in (axes or self._AXIS_ORDERS)}
 d.update(kwargs)
 return d

@classmethod
def _construct_axes_from_arguments(
 cls, args, kwargs, require_all: bool = False, sentinel=None
):
 """
 Construct and returns axes if supplied in args/kwargs.

 If require_all, raise if all axis arguments are not supplied
 return a tuple of (axes, kwargs).

 sentinel specifies the default parameter when an axis is not
 supplied; useful to distinguish when a user explicitly passes None
 in scenarios where None has special meaning.
 """
 # construct the args
 args = list(args)
 for a in cls._AXIS_ORDERS:

 # look for a argument by position
 if a not in kwargs:
 try:
 kwargs[a] = args.pop(0)
 except IndexError as err:
 if require_all:
 raise TypeError(
 "not enough/duplicate arguments specified!"
) from err

 axes = {a: kwargs.pop(a, sentinel) for a in cls._AXIS_ORDERS}
 return axes, kwargs

@classmethod
def _get_axis_number(cls, axis: Axis) -> int:
 try:
 return cls._AXIS_TO_AXIS_NUMBER[axis]
 except KeyError:

```

```

 raise ValueError(f"No axis named {axis} for object type {cls.__name__}")

@classmethod
def _get_axis_name(cls, axis: Axis) -> str:
 axis_number = cls._get_axis_number(axis)
 return cls._AXIS_ORDERS[axis_number]

def _get_axis(self, axis: Axis) -> Index:
 axis_number = self._get_axis_number(axis)
 assert axis_number in {0, 1}
 return self.index if axis_number == 0 else self.columns

@classmethod
def _get_block_manager_axis(cls, axis: Axis) -> int:
 """Map the axis to the block_manager axis."""
 axis = cls._get_axis_number(axis)
 if cls._AXIS_REVERSED:
 m = cls._AXIS_LEN - 1
 return m - axis
 return axis

def _get_axis_resolvers(self, axis: str) -> Dict[str, ABCSeries]:
 # index or columns
 axis_index = getattr(self, axis)
 d = dict()
 prefix = axis[0]

 for i, name in enumerate(axis_index.names):
 if name is not None:
 key = level = name
 else:
 # prefix with 'i' or 'c' depending on the input axis
 # e.g., you must do illevel_0 for the 0th level of an unnamed
 # multiiindex
 key = f"{prefix}level_{i}"
 level = i

 level_values = axis_index.get_level_values(level)
 s = level_values.to_series()
 s.index = axis_index
 d[key] = s

 # put the index/columns itself in the dict
 if isinstance(axis_index, MultiIndex):
 dindex = axis_index
 else:
 dindex = axis_index.to_series()

 d[axis] = dindex
 return d

def _get_index_resolvers(self) -> Dict[str, ABCSeries]:
 from pandas.core.computation.parsing import clean_column_name

 d: Dict[str, ABCSeries] = {}
 for axis_name in self._AXIS_ORDERS:
 d.update(self._get_axis_resolvers(axis_name))

 return {clean_column_name(k): v for k, v in d.items() if not isinstance(k, int)}

def _get_cleaned_column_resolvers(self) -> Dict[str, ABCSeries]:
 """
 Return the special character free column resolvers of a dataframe.

```

```
Column names with special characters are 'cleaned up' so that they can
be referred to by backtick quoting.
Used in :meth:`DataFrame.eval`.
"""
from pandas.core.computation.parsing import clean_column_name

if isinstance(self, ABCSeries):
 return {clean_column_name(self.name): self}

return {
 clean_column_name(k): v for k, v in self.items() if not isinstance(k, int)
}

@property
def _info_axis(self) -> Index:
 return getattr(self, self._info_axis_name)

@property
def _stat_axis(self) -> Index:
 return getattr(self, self._stat_axis_name)

@property
def shape(self) -> Tuple[int, ...]:
 """
 Return a tuple of axis dimensions
 """
 return tuple(len(self._get_axis(a)) for a in self._AXIS_ORDERS)

@property
def axes(self) -> List[Index]:
 """
 Return index label(s) of the internal NDFrame
 """
 # we do it this way because if we have reversed axes, then
 # the block manager shows them reversed
 return [self._get_axis(a) for a in self._AXIS_ORDERS]

@property
def ndim(self) -> int:
 """
 Return an int representing the number of axes / array dimensions.

 Return 1 if Series. Otherwise return 2 if DataFrame.
 See Also

 ndarray.ndim : Number of array dimensions.

 Examples

 >>> s = pd.Series({'a': 1, 'b': 2, 'c': 3})
 >>> s.ndim
 1

 >>> df = pd.DataFrame({'col1': [1, 2], 'col2': [3, 4]})
 >>> df.ndim
 2
 """
 return self._mgr.ndim

@property
def size(self) -> int:
```

```

"""
Return an int representing the number of elements in this object.

Return the number of rows if Series. Otherwise return the number of
rows times number of columns if DataFrame.

See Also

ndarray.size : Number of elements in the array.

Examples

>>> s = pd.Series({'a': 1, 'b': 2, 'c': 3})
>>> s.size
3

>>> df = pd.DataFrame({'col1': [1, 2], 'col2': [3, 4]})
>>> df.size
4
"""
return np.prod(self.shape)

@property
def _selected_obj(self: FrameOrSeries) -> FrameOrSeries:
 """ internal compat with SelectionMixin """
 return self

@property
def _obj_with_exclusions(self: FrameOrSeries) -> FrameOrSeries:
 """ internal compat with SelectionMixin """
 return self

def set_axis(self, labels, axis: Axis = 0, inplace: bool = False):
 """
 Assign desired index to given axis.

 Indexes for%(extended_summary_sub)s row labels can be changed by assigning
 a list-like or Index.

 Parameters

 labels : list-like, Index
 The values for the new index.

 axis : %(axes_single_arg)s, default 0
 The axis to update. The value 0 identifies the rows%(axis_description_sub)s.

 inplace : bool, default False
 Whether to return a new %(klass)s instance.

 Returns

 renamed : %(klass)s or None
 An object of type %(klass)s if inplace=False, None otherwise.

 See Also

 %(klass)s.rename_axis : Alter the name of the index%(see_also_sub)s.
 """
 if inplace:
 setattr(self, self._get_axis_name(axis), labels)
 else:
 obj = self.copy()

```

```

 obj.set_axis(labels, axis=axis, inplace=True)
 return obj

 def _set_axis(self, axis: int, labels: Index) -> None:
 labels = ensure_index(labels)
 self._mgr.set_axis(axis, labels)
 self._clear_item_cache()

 def swapaxes(self: FrameOrSeries, axis1, axis2, copy=True) -> FrameOrSeries:
 """
 Interchange axes and swap values axes appropriately.

 Returns

 y : same as input
 """
 i = self._get_axis_number(axis1)
 j = self._get_axis_number(axis2)

 if i == j:
 if copy:
 return self.copy()
 return self

 mapping = {i: j, j: i}

 new_axes = (self._get_axis(mapping.get(k, k)) for k in range(self._AXIS_LEN))
 new_values = self.values.swapaxes(i, j)
 if copy:
 new_values = new_values.copy()

 # ignore needed because of NDFrame constructor is different than
 # DataFrame/Series constructors.
 return self._constructor(new_values, *new_axes).__finalize__(# type: ignore
 self, method="swapaxes"
)

 def droplevel(self: FrameOrSeries, level, axis=0) -> FrameOrSeries:
 """
 Return DataFrame with requested index / column level(s) removed.

 .. versionadded:: 0.24.0

 Parameters

 level : int, str, or list-like
 If a string is given, must be the name of a level
 If list-like, elements must be names or positional indexes
 of levels.

 axis : {0 or 'index', 1 or 'columns'}, default 0
 Axis along which the level(s) is removed:
 * 0 or 'index': remove level(s) in column.
 * 1 or 'columns': remove level(s) in row.

 Returns

 DataFrame
 DataFrame with requested index / column level(s) removed.

 Examples

```

```

>>> df = pd.DataFrame([
... [1, 2, 3, 4],
... [5, 6, 7, 8],
... [9, 10, 11, 12]
...]).set_index([0, 1]).rename_axis(['a', 'b'])

>>> df.columns = pd.MultiIndex.from_tuples([
... ('c', 'e'), ('d', 'f')
...], names=['level_1', 'level_2'])

>>> df
level_1 c d
level_2 e f
a b
1 2 3 4
5 6 7 8
9 10 11 12

>>> df.droplevel('a')
level_1 c d
level_2 e f
b
2 3 4
6 7 8
10 11 12

>>> df.droplevel('level_2', axis=1)
level_1 c d
a b
1 2 3 4
5 6 7 8
9 10 11 12
"""

labels = self._get_axis(axis)
new_labels = labels.droplevel(level)
result = self.set_axis(new_labels, axis=axis, inplace=False)
return result

def pop(self: FrameOrSeries, item) -> FrameOrSeries:
 """
 Return item and drop from frame. Raise KeyError if not found.

 Parameters

 item : str
 Label of column to be popped.

 Returns

 Series

 Examples

 >>> df = pd.DataFrame([('falcon', 'bird', 389.0),
... ('parrot', 'bird', 24.0),
... ('lion', 'mammal', 80.5),
... ('monkey', 'mammal', np.nan)],
... columns=('name', 'class', 'max_speed'))
 >>> df
 name class max_speed
0 falcon bird 389.0
1 parrot bird 24.0
2 lion mammal 80.5

```

```
3 monkey mammal NaN

>>> df.pop('class')
0 bird
1 bird
2 mammal
3 mammal
Name: class, dtype: object

>>> df
 name max_speed
0 falcon 389.0
1 parrot 24.0
2 lion 80.5
3 monkey NaN
"""

result = self[item]
del self[item]
if self.ndim == 2:
 result._reset_cacher()

return result

def squeeze(self, axis=None):
 """
 Squeeze 1 dimensional axis objects into scalars.

 Series or DataFrames with a single element are squeezed to a scalar.
 DataFrames with a single column or a single row are squeezed to a
 Series. Otherwise the object is unchanged.

 This method is most useful when you don't know if your
 object is a Series or DataFrame, but you do know it has just a single
 column. In that case you can safely call `squeeze` to ensure you have a
 Series.

 Parameters

 axis : {0 or 'index', 1 or 'columns', None}, default None
 A specific axis to squeeze. By default, all length-1 axes are
 squeezed.

 Returns

 DataFrame, Series, or scalar
 The projection after squeezing `axis` or all the axes.

 See Also

 Series.iloc : Integer-location based indexing for selecting scalars.
 DataFrame.iloc : Integer-location based indexing for selecting Series.
 Series.to_frame : Inverse of DataFrame.squeeze for a
 single-column DataFrame.

 Examples

 >>> primes = pd.Series([2, 3, 5, 7])

 Slicing might produce a Series with a single value:

 >>> even_primes = primes[primes % 2 == 0]
 >>> even_primes
 0 2
```

```
dtype: int64

>>> even_primes.squeeze()
2

Squeezing objects with more than one value in every axis does nothing:

>>> odd_primes = primes[primes % 2 == 1]
>>> odd_primes
1 3
2 5
3 7
dtype: int64

>>> odd_primes.squeeze()
1 3
2 5
3 7
dtype: int64
```

Squeezing is even more effective when used with DataFrames.

```
>>> df = pd.DataFrame([[1, 2], [3, 4]], columns=['a', 'b'])
>>> df
 a b
0 1 2
1 3 4
```

Slicing a single column will produce a DataFrame with the columns having only one value:

```
>>> df_a = df[['a']]
>>> df_a
 a
0 1
1 3
```

So the columns can be squeezed down, resulting in a Series:

```
>>> df_a.squeeze('columns')
0 1
1 3
Name: a, dtype: int64
```

Slicing a single row from a single column will produce a single scalar DataFrame:

```
>>> df_0a = df.loc[df.index < 1, ['a']]
>>> df_0a
 a
0 1
```

Squeezing the rows produces a single scalar Series:

```
>>> df_0a.squeeze('rows')
a 1
Name: 0, dtype: int64
```

Squeezing all axes will project directly into a scalar:

```
>>> df_0a.squeeze()
1
"""
```

```

axis = range(self._AXIS_LEN) if axis is None else (self._get_axis_number(axis),)
return self.iloc[
 tuple(
 0 if i in axis and len(a) == 1 else slice(None)
 for i, a in enumerate(self.axes)
)
]

Rename

def rename(
 self: FrameOrSeries,
 mapper: Optional[Renamer] = None,
 *,
 index: Optional[Renamer] = None,
 columns: Optional[Renamer] = None,
 axis: Optional[Axis] = None,
 copy: bool = True,
 inplace: bool = False,
 level: Optional[Level] = None,
 errors: str = "ignore",
) -> Optional[FrameOrSeries]:
 """
 Alter axes input function or functions. Function / dict values must be
 unique (1-to-1). Labels not contained in a dict / Series will be left
 as-is. Extra labels listed don't throw an error. Alternatively, change
 ``Series.name`` with a scalar value (Series only).

 Parameters

 %(axes)s : scalar, list-like, dict-like or function, optional
 Scalar or list-like will alter the ``Series.name`` attribute,
 and raise on DataFrame.
 dict-like or functions are transformations to apply to
 that axis' values
 copy : bool, default True
 Also copy underlying data.
 inplace : bool, default False
 Whether to return a new %(klass)s. If True then value of copy is
 ignored.
 level : int or level name, default None
 In case of a MultiIndex, only rename labels in the specified
 level.
 errors : {'ignore', 'raise'}, default 'ignore'
 If 'raise', raise a `KeyError` when a dict-like `mapper`, `index`,
 or `columns` contains labels that are not present in the Index
 being transformed.
 If 'ignore', existing keys will be renamed and extra keys will be
 ignored.

 Returns

 renamed : %(klass)s (new object)

 Raises

 KeyError
 If any of the labels is not found in the selected axis and
 "errors='raise'".

 See Also

```

```
NDFrame.rename_axis

Examples

>>> s = pd.Series([1, 2, 3])
>>> s
0 1
1 2
2 3
dtype: int64
>>> s.rename("my_name") # scalar, changes Series.name
0 1
1 2
2 3
Name: my_name, dtype: int64
>>> s.rename(lambda x: x ** 2) # function, changes labels
0 1
1 2
4 3
dtype: int64
>>> s.rename({1: 3, 2: 5}) # mapping, changes labels
0 1
3 2
5 3
dtype: int64
```

Since ``DataFrame`` doesn't have a ``.name`` attribute, only mapping-type arguments are allowed.

```
>>> df = pd.DataFrame({"A": [1, 2, 3], "B": [4, 5, 6]})
>>> df.rename(2)
Traceback (most recent call last):
...
TypeError: 'int' object is not callable
```

``DataFrame.rename`` supports two calling conventions

- \* ``(index=index\_mapper, columns=columns\_mapper, ...)``
- \* ``(mapper, axis={'index', 'columns'}, ...)``

We **highly** recommend using keyword arguments to clarify your intent.

```
>>> df.rename(index=str, columns={"A": "a", "B": "c"})
 a c
0 1 4
1 2 5
2 3 6

>>> df.rename(index=str, columns={"A": "a", "C": "c"})
 a B
0 1 4
1 2 5
2 3 6
```

Using axis-style parameters

```
>>> df.rename(str.lower, axis='columns')
 a b
0 1 4
1 2 5
2 3 6
```

```

>>> df.rename({1: 2, 2: 4}, axis='index')
 A B
0 1 4
2 2 5
4 3 6

See the :ref:`user guide <basics.rename>` for more.
"""
if mapper is None and index is None and columns is None:
 raise TypeError("must pass an index to rename")

if index is not None or columns is not None:
 if axis is not None:
 raise TypeError(
 "Cannot specify both 'axis' and any of 'index' or 'columns'"
)
 elif mapper is not None:
 raise TypeError(
 "Cannot specify both 'mapper' and any of 'index' or 'columns'"
)
else:
 # use the mapper argument
 if axis and self._get_axis_number(axis) == 1:
 columns = mapper
 else:
 index = mapper

result = self if inplace else self.copy(deep=copy)

for axis_no, replacements in enumerate((index, columns)):
 if replacements is None:
 continue

 ax = self._get_axis(axis_no)
 f = com.get_rename_function(replacements)

 if level is not None:
 level = ax._get_level_number(level)

 # GH 13473
 if not callable(replacements):
 indexer = ax.get_indexer_for(replacements)
 if errors == "raise" and len(indexer[indexer == -1]):
 missing_labels = [
 label
 for index, label in enumerate(replacements)
 if indexer[index] == -1
]
 raise KeyError(f"{missing_labels} not found in axis")

 new_index = ax._transform_index(f, level)
 result.set_axis(new_index, axis=axis_no, inplace=True)
 result._clear_item_cache()

if inplace:
 self._update_inplace(result)
 return None
else:
 return result.__finalize__(self, method="rename")

@rewrite_axis_style_signature("mapper", [("copy", True), ("inplace", False)])
def rename_axis(self, mapper=lib.no_default, **kwargs):
 """

```

```
Set the name of the axis for the index or columns.

Parameters

mapper : scalar, list-like, optional
 Value to set the axis name attribute.
index, columns : scalar, list-like, dict-like or function, optional
 A scalar, list-like, dict-like or functions transformations to
 apply to that axis' values.

 Use either ``mapper`` and ``axis`` to
 specify the axis to target with ``mapper``, or ``index``
 and/or ``columns``.

.. versionchanged:: 0.24.0

axis : {0 or 'index', 1 or 'columns'}, default 0
 The axis to rename.
copy : bool, default True
 Also copy underlying data.
inplace : bool, default False
 Modifies the object directly, instead of creating a new Series
 or DataFrame.

Returns

Series, DataFrame, or None
 The same type as the caller or None if `inplace` is True.

See Also

Series.rename : Alter Series index labels or name.
DataFrame.rename : Alter DataFrame index labels or name.
Index.rename : Set new names on index.

Notes

``DataFrame.rename_axis`` supports two calling conventions

* ``(``index=index_mapper, columns=columns_mapper, ...)``*
* ``(``mapper, axis={'index', 'columns'}, ...)````

The first calling convention will only modify the names of
the index and/or the names of the Index object that is the columns.
In this case, the parameter ``copy`` is ignored.

The second calling convention will modify the names of the
the corresponding index if mapper is a list or a scalar.
However, if mapper is dict-like or a function, it will use the
deprecated behavior of modifying the axis *labels*.

We *highly* recommend using keyword arguments to clarify your
intent.

Examples

Series

>>> s = pd.Series(["dog", "cat", "monkey"])
>>> s
0 dog
1 cat
2 monkey
```

```

dtype: object
>>> s.rename_axis("animal")
animal
0 dog
1 cat
2 monkey
dtype: object

DataFrame

>>> df = pd.DataFrame({"num_legs": [4, 4, 2],
... "num_arms": [0, 0, 2]},
... ["dog", "cat", "monkey"])
>>> df
 num_legs num_arms
dog 4 0
cat 4 0
monkey 2 2
>>> df = df.rename_axis("animal")
>>> df
 num_legs num_arms
animal
dog 4 0
cat 4 0
monkey 2 2
>>> df = df.rename_axis("limbs", axis="columns")
>>> df
limbs num_legs num_arms
animal
dog 4 0
cat 4 0
monkey 2 2

MultiIndex

>>> df.index = pd.MultiIndex.from_product([['mammal'],
... ['dog', 'cat', 'monkey']],
... names=['type', 'name'])
>>> df
limbs num_legs num_arms
type name
mammal dog 4 0
 cat 4 0
 monkey 2 2

>>> df.rename_axis(index={'type': 'class'})
limbs num_legs num_arms
class name
mammal dog 4 0
 cat 4 0
 monkey 2 2

>>> df.rename_axis(columns=str.upper)
LIMBS num_legs num_arms
type name
mammal dog 4 0
 cat 4 0
 monkey 2 2
"""
axes, kwargs = self._construct_axes_from_arguments(
 (), kwargs, sentinel=lib.no_default
)
copy = kwargs.pop("copy", True)

```

```

inplace = kwargs.pop("inplace", False)
axis = kwargs.pop("axis", 0)
if axis is not None:
 axis = self._get_axis_number(axis)

if kwargs:
 raise TypeError(
 "rename_axis() got an unexpected keyword "
 f'argument "{list(kwargs.keys())[0]}")'
)

inplace = validate_bool_kwarg(inplace, "inplace")

if mapper is not lib.no_default:
 # Use v0.23 behavior if a scalar or list
 non_mapper = is_scalar(mapper) or (
 is_list_like(mapper) and not is_dict_like(mapper)
)
 if non_mapper:
 return self._set_axis_name(mapper, axis=axis, inplace=inplace)
 else:
 raise ValueError("Use `rename` to alter labels with a mapper.")
else:
 # Use new behavior. Means that index and/or columns
 # is specified
 result = self if inplace else self.copy(deep=copy)

 for axis in range(self._AXIS_LEN):
 v = axes.get(self._get_axis_name(axis))
 if v is lib.no_default:
 continue
 non_mapper = is_scalar(v) or (is_list_like(v) and not is_dict_like(v))
 if non_mapper:
 newnames = v
 else:
 f = com.get_rename_function(v)
 curnames = self._get_axis(axis).names
 newnames = [f(name) for name in curnames]
 result._set_axis_name(newnames, axis=axis, inplace=True)
 if not inplace:
 return result

def _set_axis_name(self, name, axis=0, inplace=False):
 """
 Set the name(s) of the axis.

 Parameters

 name : str or list of str
 Name(s) to set.
 axis : {0 or 'index', 1 or 'columns'}, default 0
 The axis to set the label. The value 0 or 'index' specifies index,
 and the value 1 or 'columns' specifies columns.
 inplace : bool, default False
 If `True`, do operation inplace and return None.

 Returns

 Series, DataFrame, or None
 The same type as the caller or `None` if `inplace` is `True`.
 """

 See Also

```

```

DataFrame.rename : Alter the axis labels of :class:`DataFrame`.
Series.rename : Alter the index labels or set the index name
 of :class:`Series`.
Index.rename : Set the name of :class:`Index` or :class:`MultiIndex`.

Examples

>>> df = pd.DataFrame({"num_legs": [4, 4, 2]},
... ["dog", "cat", "monkey"])
>>> df
 num_legs
dog 4
cat 4
monkey 2
>>> df._set_axis_name("animal")
 num_legs
animal
dog 4
cat 4
monkey 2
>>> df.index = pd.MultiIndex.from_product(
... [["mammal"], ['dog', 'cat', 'monkey']])
>>> df._set_axis_name(["type", "name"])
 num_legs
type name
mammal dog 4
 cat 4
 monkey 2
"""
axis = self._get_axis_number(axis)
idx = self._get_axis(axis).set_names(name)

inplace = validate_bool_kwarg(inplace, "inplace")
renamed = self if inplace else self.copy()
renamed.set_axis(idx, axis=axis, inplace=True)
if not inplace:
 return renamed

Comparison Methods

def _indexed_same(self, other) -> bool:
 return all(
 self._get_axis(a).equals(other._get_axis(a)) for a in self._AXIS_ORDERS
)

def equals(self, other):
 """
 Test whether two objects contain the same elements.

 This function allows two Series or DataFrames to be compared against
 each other to see if they have the same shape and elements. NaNs in
 the same location are considered equal. The column headers do not
 need to have the same type, but the elements within the columns must
 be the same dtype.

 Parameters

 other : Series or DataFrame
 The other Series or DataFrame to be compared with the first.

 Returns

```

```
bool
 True if all elements are the same in both objects, False
 otherwise.

See Also

Series.eq : Compare two Series objects of the same length
 and return a Series where each element is True if the element
 in each Series is equal, False otherwise.
DataFrame.eq : Compare two DataFrame objects of the same shape and
 return a DataFrame where each element is True if the respective
 element in each DataFrame is equal, False otherwise.
testing.assert_series_equal : Raises an AssertionError if left and
 right are not equal. Provides an easy interface to ignore
 inequality in dtypes, indexes and precision among others.
testing.assert_frame_equal : Like assert_series_equal, but targets
 DataFrames.
numpy.array_equal : Return True if two arrays have the same shape
 and elements, False otherwise.
```

## Notes

```

This function requires that the elements have the same dtype as their
respective elements in the other Series or DataFrame. However, the
column labels do not need to have the same type, as long as they are
still considered equal.
```

## Examples

```

>>> df = pd.DataFrame({1: [10], 2: [20]})
>>> df
 1 2
0 10 20
```

DataFrames df and exactly\_equal have the same types and values for
their elements and column labels, which will return True.

```
>>> exactly_equal = pd.DataFrame({1: [10], 2: [20]})
>>> exactly_equal
 1 2
0 10 20
>>> df.equals(exactly_equal)
True
```

DataFrames df and different\_column\_type have the same element
types and values, but have different types for the column labels,
which will still return True.

```
>>> different_column_type = pd.DataFrame({1.0: [10], 2.0: [20]})
>>> different_column_type
 1.0 2.0
0 10 20
>>> df.equals(different_column_type)
True
```

DataFrames df and different\_data\_type have different types for the
same values for their elements, and will return False even though
their column labels are the same values and types.

```
>>> different_data_type = pd.DataFrame({1: [10.0], 2: [20.0]})
>>> different_data_type
 1 2
0 10.0 20.0
```

```

>>> df.equals(different_data_type)
False
"""
if not isinstance(other, self._constructor):
 return False
return self._mgr.equals(other._mgr)

Unary Methods

def __neg__(self):
 values = self._values
 if is_bool_dtype(values):
 arr = operator.inv(values)
 elif (
 is_numeric_dtype(values)
 or is_timedelta64_dtype(values)
 or is_object_dtype(values)
):
 arr = operator.neg(values)
 else:
 raise TypeError(f"Unary negative expects numeric dtype, not {values.dtype}")
 return self.__array_wrap__(arr)

def __pos__(self):
 values = self._values
 if is_bool_dtype(values):
 arr = values
 elif (
 is_numeric_dtype(values)
 or is_timedelta64_dtype(values)
 or is_object_dtype(values)
):
 arr = operator.pos(values)
 else:
 raise TypeError(f"Unary plus expects numeric dtype, not {values.dtype}")
 return self.__array_wrap__(arr)

def __invert__(self):
 if not self.size:
 # inv fails with 0 len
 return self

 new_data = self._mgr.apply(operator.invert)
 result = self._constructor(new_data).__finalize__(self, method="__invert__")
 return result

def __nonzero__(self):
 raise ValueError(
 f"The truth value of a {type(self).__name__} is ambiguous. "
 "Use a.empty, a.bool(), a.item(), a.any() or a.all()."
)

__bool__ = __nonzero__

def bool(self):
 """
 Return the bool of a single element PandasObject.

 This must be a boolean scalar value, either True or False. Raise a
 ValueError if the PandasObject does not have exactly 1 element, or that
 element is not boolean

```

```

>Returns

bool
 Same single boolean value converted to bool type.
"""
v = self.squeeze()
if isinstance(v, (bool, np.bool_)):
 return bool(v)
elif is_scalar(v):
 raise ValueError(
 "bool cannot act on a non-boolean single element "
 f"{type(self).__name__}"
)

self.__nonzero__()

def __abs__(self: FrameOrSeries) -> FrameOrSeries:
 return self.abs()

def __round__(self: FrameOrSeries, decimals: int = 0) -> FrameOrSeries:
 return self.round(decimals)

Label or Level Combination Helpers
#
A collection of helper methods for DataFrame/Series operations that
accept a combination of column/index labels and levels. All such
operations should utilize/extend these methods when possible so that we
have consistent precedence and validation logic throughout the library.

def _is_level_reference(self, key, axis=0):
 """
 Test whether a key is a level reference for a given axis.

 To be considered a level reference, `key` must be a string that:
 - (axis=0): Matches the name of an index level and does NOT match
 a column label.
 - (axis=1): Matches the name of a column level and does NOT match
 an index label.

 Parameters

 key : str
 Potential level name for the given axis
 axis : int, default 0
 Axis that levels are associated with (0 for index, 1 for columns)

 Returns

 is_level : bool
 """
 axis = self._get_axis_number(axis)

 return (
 key is not None
 and is_hashable(key)
 and key in self.axes[axis].names
 and not self._is_label_reference(key, axis=axis)
)

def _is_label_reference(self, key, axis=0) -> bool_t:
 """
 Test whether a key is a label reference for a given axis.

```

```

To be considered a label reference, `key` must be a string that:
- (axis=0): Matches a column label
- (axis=1): Matches an index label

Parameters

key: str
 Potential label name
axis: int, default 0
 Axis perpendicular to the axis that labels are associated with
 (0 means search for column labels, 1 means search for index labels)

Returns

is_label: bool
"""
axis = self._get_axis_number(axis)
other_axes = (ax for ax in range(self._AXIS_LEN) if ax != axis)

return (
 key is not None
 and is_hashable(key)
 and any(key in self.axes[ax] for ax in other_axes)
)

def _is_label_or_level_reference(self, key: str, axis: int = 0) -> bool_t:
"""
Test whether a key is a label or level reference for a given axis.

To be considered either a label or a level reference, `key` must be a
string that:
- (axis=0): Matches a column label or an index level
- (axis=1): Matches an index label or a column level

Parameters

key: str
 Potential label or level name
axis: int, default 0
 Axis that levels are associated with (0 for index, 1 for columns)

Returns

is_label_or_level: bool
"""
return self._is_level_reference(key, axis=axis) or self._is_label_reference(
 key, axis=axis
)

def _check_label_or_level_ambiguity(self, key, axis: int = 0) -> None:
"""
Check whether `key` is ambiguous.

By ambiguous, we mean that it matches both a level of the input
`axis` and a label of the other axis.

Parameters

key: str or object
 Label or level name.
axis: int, default 0
 Axis that levels are associated with (0 for index, 1 for columns).

```

```

Raises

ValueError: `key` is ambiguous
"""
axis = self._get_axis_number(axis)
other_axes = (ax for ax in range(self._AXIS_LEN) if ax != axis)

if (
 key is not None
 and is_hashable(key)
 and key in self.axes[axis].names
 and any(key in self.axes[ax] for ax in other_axes)
):
 # Build an informative and grammatical warning
 level_article, level_type = (
 ("an", "index") if axis == 0 else ("a", "column")
)

 label_article, label_type = (
 ("a", "column") if axis == 0 else ("an", "index")
)

 msg = (
 f"'{key}' is both {level_article} {level_type} level and "
 f"{label_article} {label_type} label, which is ambiguous."
)
 raise ValueError(msg)

def _get_label_or_level_values(self, key: str, axis: int = 0) -> np.ndarray:
 """
 Return a 1-D array of values associated with `key`, a label or level
 from the given `axis`.

 Retrieval logic:
 - (axis=0): Return column values if `key` matches a column label.
 Otherwise return index level values if `key` matches an index
 level.
 - (axis=1): Return row values if `key` matches an index label.
 Otherwise return column level values if 'key' matches a column
 level

 Parameters

 key: str
 Label or level name.
 axis: int, default 0
 Axis that levels are associated with (0 for index, 1 for columns)

 Returns

 values: np.ndarray

 Raises

 KeyError
 if `key` matches neither a label nor a level
 ValueError
 if `key` matches multiple labels
 FutureWarning
 if `key` is ambiguous. This will become an ambiguity error in a
 future version

```

```

"""
axis = self._get_axis_number(axis)
other_axes = [ax for ax in range(self._AXIS_LEN) if ax != axis]

if self._is_label_reference(key, axis=axis):
 self._check_label_or_level_ambiguity(key, axis=axis)
 values = self.xs(key, axis=other_axes[0])._values
elif self._is_level_reference(key, axis=axis):
 values = self.axes[axis].get_level_values(key)._values
else:
 raise KeyError(key)

Check for duplicates
if values.ndim > 1:

 if other_axes and isinstance(self._get_axis(other_axes[0]), MultiIndex):
 multi_message = (
 "\n"
 "For a multi-index, the label must be a "
 "tuple with elements corresponding to each level."
)
 else:
 multi_message = ""

 label_axis_name = "column" if axis == 0 else "index"
 raise ValueError(
 (
 f"The {label_axis_name} label '{key}' "
 f"is not unique.{multi_message}"
)
)

return values

def _drop_labels_or_levels(self, keys, axis: int = 0):
"""
Drop labels and/or levels for the given `axis`.

For each key in `keys`:
- (axis=0): If key matches a column label then drop the column.
 Otherwise if key matches an index level then drop the level.
- (axis=1): If key matches an index label then drop the row.
 Otherwise if key matches a column level then drop the level.

Parameters

keys: str or list of str
 labels or levels to drop
axis: int, default 0
 Axis that levels are associated with (0 for index, 1 for columns)

Returns

dropped: DataFrame

Raises

ValueError
 if any `keys` match neither a label nor a level
"""
axis = self._get_axis_number(axis)

Validate keys

```

```

keys = com.maybe_make_list(keys)
invalid_keys = [
 k for k in keys if not self._is_label_or_level_reference(k, axis=axis)
]

if invalid_keys:
 raise ValueError(
 (
 "The following keys are not valid labels or "
 f"levels for axis {axis}: {invalid_keys}"
)
)

Compute levels and labels to drop
levels_to_drop = [k for k in keys if self._is_level_reference(k, axis=axis)]

labels_to_drop = [k for k in keys if not self._is_level_reference(k, axis=axis)]

Perform copy upfront and then use inplace operations below.
This ensures that we always perform exactly one copy.
``copy`` and/or ``inplace`` options could be added in the future.
dropped = self.copy()

if axis == 0:
 # Handle dropping index levels
 if levels_to_drop:
 dropped.reset_index(levels_to_drop, drop=True, inplace=True)

 # Handle dropping columns labels
 if labels_to_drop:
 dropped.drop(labels_to_drop, axis=1, inplace=True)
else:
 # Handle dropping column levels
 if levels_to_drop:
 if isinstance(dropped.columns, MultiIndex):
 # Drop the specified levels from the MultiIndex
 dropped.columns = dropped.columns.droplevel(levels_to_drop)
 else:
 # Drop the last level of Index by replacing with
 # a RangeIndex
 dropped.columns = RangeIndex(dropped.columns.size)

 # Handle dropping index labels
 if labels_to_drop:
 dropped.drop(labels_to_drop, axis=0, inplace=True)

return dropped

Iteration

def __hash__(self):
 raise TypeError(
 f"{repr(type(self).__name__)} objects are mutable, "
 f"thus they cannot be hashed"
)

def __iter__(self):
 """
 Iterate over info axis.

 Returns

 """

```

```
iterator
 Info axis as iterator.
"""
return iter(self._info_axis)

can we get a better explanation of this?
def keys(self):
 """
 Get the 'info axis' (see Indexing for more).

 This is index for Series, columns for DataFrame.

 Returns

 Index
 Info axis.
 """
 return self._info_axis

def items(self):
 """
 Iterate over (label, values) on info axis

 This is index for Series and columns for DataFrame.

 Returns

 Generator
 """
 for h in self._info_axis:
 yield h, self[h]

@doc(items)
def iteritems(self):
 return self.items()

def __len__(self) -> int:
 """Returns length of info axis"""
 return len(self._info_axis)

def __contains__(self, key) -> bool_t:
 """True if the key is in the info axis"""
 return key in self._info_axis

@property
def empty(self) -> bool_t:
 """
 Indicator whether DataFrame is empty.

 True if DataFrame is entirely empty (no items), meaning any of the
 axes are of length 0.

 Returns

 bool
 If DataFrame is empty, return True, if not return False.
 See Also

 Series.dropna : Return series without null values.
 DataFrame.dropna : Return DataFrame with labels on given axis omitted
 where (all or any) data are missing.

```

```
Notes
```

```

If DataFrame contains only NaNs, it is still not considered empty. See
the example below.
```

```
Examples
```

```

```

```
An example of an actual empty DataFrame. Notice the index is empty:
```

```
>>> df_empty = pd.DataFrame({'A' : []})
>>> df_empty
Empty DataFrame
Columns: [A]
Index: []
>>> df_empty.empty
True
```

```
If we only have NaNs in our DataFrame, it is not considered empty! We
will need to drop the NaNs to make the DataFrame empty:
```

```
>>> df = pd.DataFrame({'A' : [np.nan] })
>>> df
 A
0 NaN
>>> df.empty
False
>>> df.dropna().empty
True
"""
 return any(len(self._get_axis(a)) == 0 for a in self._AXIS_ORDERS)

Array Interface

This is also set in IndexOpsMixin
GH#23114 Ensure ndarray.__op__(DataFrame) returns NotImplemented
__array_priority__ = 1000

def __array__(self, dtype=None) -> np.ndarray:
 return np.asarray(self._values, dtype=dtype)

def __array_wrap__(self, result, context=None):
 result = lib.item_from_zerodim(result)
 if is_scalar(result):
 # e.g. we get here with np.ptp(series)
 # ptp also requires the item_from_zerodim
 return result
 d = self._construct_axes_dict(self._AXIS_ORDERS, copy=False)
 return self._constructor(result, **d).__finalize__(
 self, method="__array_wrap__"
)

ideally we would define this to avoid the getattr checks, but
is slower
@property
def __array_interface__(self):
""" provide numpy array interface method """
values = self.values
return dict(typestr=values.dtype.str, shape=values.shape, data=values)

Picklability
```

```

def __getstate__(self) -> Dict[str, Any]:
 meta = {k: getattr(self, k, None) for k in self._metadata}
 return dict(
 _mgr=self._mgr,
 _typ=self._typ,
 _metadata=self._metadata,
 attrs=self.attrs,
 **meta,
)

def __setstate__(self, state):
 if isinstance(state, BlockManager):
 self._mgr = state
 elif isinstance(state, dict):
 if "_data" in state and "_mgr" not in state:
 # compat for older pickles
 state["_mgr"] = state.pop("_data")
 typ = state.get("_typ")
 if typ is not None:
 attrs = state.get("_attrs", {})
 object.__setattr__(self, "_attrs", attrs)

 # set in the order of internal names
 # to avoid definitional recursion
 # e.g. say fill_value needing _mgr to be
 # defined
 meta = set(self._internal_names + self._metadata)
 for k in list(meta):
 if k in state:
 v = state[k]
 object.__setattr__(self, k, v)

 for k, v in state.items():
 if k not in meta:
 object.__setattr__(self, k, v)

 else:
 raise NotImplementedError("Pre-0.12 pickles are no longer supported")
 elif len(state) == 2:
 raise NotImplementedError("Pre-0.12 pickles are no longer supported")

 self._item_cache = {}

Rendering Methods

def __repr__(self) -> str:
 # string representation based upon iterating over self
 # (since, by definition, `PandasContainers` are iterable)
 prepr = f"[{','}.join(map(pprint_thing, self))}]"
 return f"{type(self).__name__}({prepr})"

def __repr_latex__(self):
 """
 Returns a LaTeX representation for a particular object.
 Mainly for use with nbconvert (jupyter notebook conversion to pdf).
 """
 if config.get_option("display.latex.repr"):
 return self.to_latex()
 else:
 return None

```

```
def _repr_data_resource_(self):
 """
 Not a real Jupyter special repr method, but we use the same
 naming convention.
 """
 if config.get_option("display.html.table_schema"):
 data = self.head(config.get_option("display.max_rows"))
 payload = json.loads(
 data.to_json(orient="table"), object_pairs_hook=collections.OrderedDict
)
 return payload

I/O Methods

_shared_docs[
 "to_markdown"
] = """
Print %(klass)s in Markdown-friendly format.

.. versionadded:: 1.0.0

Parameters

buf : str, Path or StringIO-like, optional, default None
 Buffer to write to. If None, the output is returned as a string.
mode : str, optional
 Mode in which file is opened.
**kwargs
 These parameters will be passed to `tabulate`.

Returns

str
 %(klass)s in Markdown-friendly format.
"""

@doc(klass="object")
def to_excel(
 self,
 excel_writer,
 sheet_name="Sheet1",
 na_rep="",
 float_format=None,
 columns=None,
 header=True,
 index=True,
 index_label=None,
 startrow=0,
 startcol=0,
 engine=None,
 merge_cells=True,
 encoding=None,
 inf_rep="inf",
 verbose=True,
 freeze_panes=None,
) -> None:
 """
 Write {klass} to an Excel sheet.

 To write a single {klass} to an Excel .xlsx file it is only necessary to
 specify a target file name. To write to multiple sheets it is necessary to
 create an `ExcelWriter` object with a target file name, and specify a sheet

```

in the file to write to.

Multiple sheets may be written to by specifying unique `sheet\_name`. With all data written to the file it is necessary to save the changes. Note that creating an `ExcelWriter` object with a file name that already exists will result in the contents of the existing file being erased.

#### Parameters

-----

`excel_writer : str or ExcelWriter object`  
File path or existing ExcelWriter.  
`sheet_name : str, default 'Sheet1'`  
Name of sheet which will contain DataFrame.  
`na_rep : str, default ''`  
Missing data representation.  
`float_format : str, optional`  
Format string for floating point numbers. For example  
``float\_format="%.2f"`` will format 0.1234 to 0.12.  
`columns : sequence or list of str, optional`  
Columns to write.  
`header : bool or list of str, default True`  
Write out the column names. If a list of string is given it is  
assumed to be aliases for the column names.  
`index : bool, default True`  
Write row names (index).  
`index_label : str or sequence, optional`  
Column label for index column(s) if desired. If not specified, and  
'header' and 'index' are True, then the index names are used. A  
sequence should be given if the DataFrame uses MultiIndex.  
`startrow : int, default 0`  
Upper left cell row to dump data frame.  
`startcol : int, default 0`  
Upper left cell column to dump data frame.  
`engine : str, optional`  
Write engine to use, 'openpyxl' or 'xlsxwriter'. You can also set this  
via the options ``io.excel.xlsx.writer``, ``io.excel.xls.writer``, and  
``io.excel.xls.writer``.  
`merge_cells : bool, default True`  
Write MultiIndex and Hierarchical Rows as merged cells.  
`encoding : str, optional`  
Encoding of the resulting excel file. Only necessary for xlwt,  
other writers support unicode natively.  
`inf_rep : str, default 'inf'`  
Representation for infinity (there is no native representation for  
infinity in Excel).  
`verbose : bool, default True`  
Display more information in the error logs.  
`freeze_panes : tuple of int (length 2), optional`  
Specifies the one-based bottommost row and rightmost column that  
is to be frozen.

#### See Also

-----

`to_csv : Write DataFrame to a comma-separated values (csv) file.`  
`ExcelWriter : Class for writing DataFrame objects into excel sheets.`  
`read_excel : Read an Excel file into a pandas DataFrame.`  
`read_csv : Read a comma-separated values (csv) file into DataFrame.`

#### Notes

-----

For compatibility with :meth:`~DataFrame.to\_csv`,  
to\_excel serializes lists and dicts to strings before writing.

Once a workbook has been saved it is not possible write further data without rewriting the whole workbook.

Examples

-----

Create, write to and save a workbook:

```
>>> df1 = pd.DataFrame([['a', 'b'], ['c', 'd']],
... index=['row 1', 'row 2'],
... columns=['col 1', 'col 2'])
>>> df1.to_excel("output.xlsx") # doctest: +SKIP
```

To specify the sheet name:

```
>>> df1.to_excel("output.xlsx",
... sheet_name='Sheet_name_1') # doctest: +SKIP
```

If you wish to write to more than one sheet in the workbook, it is necessary to specify an ExcelWriter object:

```
>>> df2 = df1.copy()
>>> with pd.ExcelWriter('output.xlsx') as writer: # doctest: +SKIP
... df1.to_excel(writer, sheet_name='Sheet_name_1')
... df2.to_excel(writer, sheet_name='Sheet_name_2')
```

ExcelWriter can also be used to append to an existing Excel file:

```
>>> with pd.ExcelWriter('output.xlsx',
... mode='a') as writer: # doctest: +SKIP
... df.to_excel(writer, sheet_name='Sheet_name_3')
```

To set the library that is used to write the Excel file, you can pass the `engine` keyword (the default engine is automatically chosen depending on the file extension):

```
>>> df1.to_excel('output1.xlsx', engine='xlsxwriter') # doctest: +SKIP
"""

df = self if isinstance(self, ABCDataFrame) else self.to_frame()

from pandas.io.formats.excel import ExcelFormatter

formatter = ExcelFormatter(
 df,
 na_rep=na_rep,
 cols=columns,
 header=header,
 float_format=float_format,
 index=index,
 index_label=index_label,
 merge_cells=merge_cells,
 inf_rep=inf_rep,
)
formatter.write(
 excel_writer,
 sheet_name=sheet_name,
 startrow=startrow,
 startcol=startcol,
 freeze_panes=freeze_panes,
 engine=engine,
)
```

```

def to_json(
 self,
 path_or_buf: Optional[FilePathOrBuffer] = None,
 orient: Optional[str] = None,
 date_format: Optional[str] = None,
 double_precision: int = 10,
 force_ascii: bool_t = True,
 date_unit: str = "ms",
 default_handler: Optional[Callable[[Any], JSONSerializable]] = None,
 lines: bool_t = False,
 compression: Optional[str] = "infer",
 index: bool_t = True,
 indent: Optional[int] = None,
) -> Optional[str]:
 """
 Convert the object to a JSON string.

 Note NaN's and None will be converted to null and datetime objects
 will be converted to UNIX timestamps.

 Parameters

 path_or_buf : str or file handle, optional
 File path or object. If not specified, the result is returned as
 a string.
 orient : str
 Indication of expected JSON string format.

 * Series:
 - default is 'index'
 - allowed values are: {'split', 'records', 'index', 'table'}.

 * DataFrame:
 - default is 'columns'
 - allowed values are: {'split', 'records', 'index', 'columns',
 'values', 'table'}.

 * The format of the JSON string:
 - 'split' : dict like {'index' -> [index], 'columns' -> [columns],
 'data' -> [values]}
 - 'records' : list like [{column -> value}, ..., {column -> value}]
 - 'index' : dict like {index -> {column -> value}}
 - 'columns' : dict like {column -> {index -> value}}
 - 'values' : just the values array
 - 'table' : dict like {'schema': {schema}, 'data': {data}}

 Describing the data, where data component is like ``orient='records'``.

 .. versionchanged:: 0.20.0

 date_format : {None, 'epoch', 'iso'}
 Type of date conversion. 'epoch' = epoch milliseconds,
 'iso' = ISO8601. The default depends on the `orient`. For
 ``orient='table'``, the default is 'iso'. For all other orients,
 the default is 'epoch'.
 double_precision : int, default 10
 The number of decimal places to use when encoding
 floating point values.
 force_ascii : bool, default True
 Force encoded string to be ASCII.

```

```
date_unit : str, default 'ms' (milliseconds)
 The time unit to encode to, governs timestamp and ISO8601
 precision. One of 's', 'ms', 'us', 'ns' for second, millisecond,
 microsecond, and nanosecond respectively.
default_handler : callable, default None
 Handler to call if object cannot otherwise be converted to a
 suitable format for JSON. Should receive a single argument which is
 the object to convert and return a serialisable object.
lines : bool, default False
 If 'orient' is 'records' write out line delimited json format. Will
 throw ValueError if incorrect 'orient' since others are not list
 like.

compression : {'infer', 'gzip', 'bz2', 'zip', 'xz', None}

 A string representing the compression to use in the output file,
 only used when the first argument is a filename. By default, the
 compression is inferred from the filename.

.. versionchanged:: 0.24.0
 'infer' option added and set to default
index : bool, default True
 Whether to include the index values in the JSON string. Not
 including the index (``index=False``) is only supported when
 orient is 'split' or 'table'.

.. versionadded:: 0.23.0

indent : int, optional
 Length of whitespace used to indent each record.

.. versionadded:: 1.0.0

>Returns

None or str
 If path_or_buf is None, returns the resulting json format as a
 string. Otherwise returns None.

See Also

read_json : Convert a JSON string to pandas object.

Notes

The behavior of ``indent=0`` varies from the stdlib, which does not
indent the output but does insert newlines. Currently, ``indent=0``
and the default ``indent=None`` are equivalent in pandas, though this
may change in a future release.

Examples

>>> import json
>>> df = pd.DataFrame(
... [["a", "b"], ["c", "d"]],
... index=["row 1", "row 2"],
... columns=["col 1", "col 2"],
...)
>>> result = df.to_json(orient="split")
>>> parsed = json.loads(result)
>>> json.dumps(parsed, indent=4) # doctest: +SKIP
{
```

```
"columns": [
 "col 1",
 "col 2"
],
"index": [
 "row 1",
 "row 2"
],
"data": [
 [
 [
 "a",
 "b"
],
 [
 [
 "c",
 "d"
]
]
]
}
```

Encoding/decoding a Dataframe using ``'records'`` formatted JSON.  
Note that index labels are not preserved with this encoding.

```
>>> result = df.to_json(orient="records")
>>> parsed = json.loads(result)
>>> json.dumps(parsed, indent=4) # doctest: +SKIP
[
 {
 "col 1": "a",
 "col 2": "b"
 },
 {
 "col 1": "c",
 "col 2": "d"
 }
]
```

Encoding/decoding a Dataframe using ``'index'`` formatted JSON:

```
>>> result = df.to_json(orient="index")
>>> parsed = json.loads(result)
>>> json.dumps(parsed, indent=4) # doctest: +SKIP
{
 "row 1": {
 "col 1": "a",
 "col 2": "b"
 },
 "row 2": {
 "col 1": "c",
 "col 2": "d"
 }
}
```

Encoding/decoding a Dataframe using ``'columns'`` formatted JSON:

```
>>> result = df.to_json(orient="columns")
>>> parsed = json.loads(result)
>>> json.dumps(parsed, indent=4) # doctest: +SKIP
{
 "col 1": {
 "row 1": "a",
 "row 2": "c"
 },
}
```

```
"col 2": {
 "row 1": "b",
 "row 2": "d"
}
}
```

Encoding/decoding a Dataframe using ``'values'`` formatted JSON:

```
>>> result = df.to_json(orient="values")
>>> parsed = json.loads(result)
>>> json.dumps(parsed, indent=4) # doctest: +SKIP
[
 [
 [
 "a",
 "b"
],
 [
 "c",
 "d"
]
]
```

Encoding with Table Schema:

```
>>> result = df.to_json(orient="table")
>>> parsed = json.loads(result)
>>> json.dumps(parsed, indent=4) # doctest: +SKIP
{
 "schema": {
 "fields": [
 {
 "name": "index",
 "type": "string"
 },
 {
 "name": "col 1",
 "type": "string"
 },
 {
 "name": "col 2",
 "type": "string"
 }
],
 "primaryKey": [
 "index"
],
 "pandas_version": "0.20.0"
 },
 "data": [
 {
 "index": "row 1",
 "col 1": "a",
 "col 2": "b"
 },
 {
 "index": "row 2",
 "col 1": "c",
 "col 2": "d"
 }
]
}
"""
from pandas.io import json
```

```

 if date_format is None and orient == "table":
 date_format = "iso"
 elif date_format is None:
 date_format = "epoch"

 config.is_nonnegative_int(indent)
 indent = indent or 0

 return json.to_json(
 path_or_buf=path_or_buf,
 obj=self,
 orient=orient,
 date_format=date_format,
 double_precision=double_precision,
 force_ascii=force_ascii,
 date_unit=date_unit,
 default_handler=default_handler,
 lines=lines,
 compression=compression,
 index=index,
 indent=indent,
)
}

def to_hdf(
 self,
 path_or_buf,
 key: str,
 mode: str = "a",
 complevel: Optional[int] = None,
 complib: Optional[str] = None,
 append: bool_t = False,
 format: Optional[str] = None,
 index: bool_t = True,
 min_itemsize: Optional[Union[int, Dict[str, int]]] = None,
 nan_rep=None,
 dropna: Optional[bool_t] = None,
 data_columns: Optional[Union[bool_t, List[str]]] = None,
 errors: str = "strict",
 encoding: str = "UTF-8",
) -> None:
 """
 Write the contained data to an HDF5 file using HDFStore.
 """

```

Hierarchical Data Format (HDF) is self-describing, allowing an application to interpret the structure and contents of a file with no outside information. One HDF file can hold a mix of related objects which can be accessed as a group or as individual objects.

In order to add another DataFrame or Series to an existing HDF file please use append mode and a different a key.

For more information see the :ref:`user guide <io.hdf5>`.

**Parameters**  
-----

- path\_or\_buf : str or pandas.HDFStore  
File path or HDFStore object.
- key : str  
Identifier for the group in the store.
- mode : {'a', 'w', 'r+'}, default 'a'  
Mode to open file:

```
- 'w': write, a new file is created (an existing file with
 the same name would be deleted).
- 'a': append, an existing file is opened for reading and
 writing, and if the file does not exist it is created.
- 'r+'. similar to 'a', but the file must already exist.
complevel : {0-9}, optional
 Specifies a compression level for data.
 A value of 0 disables compression.
complib : {'zlib', 'lzo', 'bzip2', 'blosc'}, default 'zlib'
 Specifies the compression library to be used.
 As of v0.20.2 these additional compressors for Blosc are supported
 (default if no compressor specified: 'blosc:blosclz'):
 {'blosc:blosclz', 'blosc:lz4', 'blosc:lz4hc', 'blosc:snappy',
 'blosc:zlib', 'blosc:zstd'}.
 Specifying a compression library which is not available issues
 a ValueError.
append : bool, default False
 For Table formats, append the input data to the existing.
format : {'fixed', 'table', None}, default 'fixed'
 Possible values:
 - 'fixed': Fixed format. Fast writing/reading. Not-appendable,
 nor searchable.
 - 'table': Table format. Write as a PyTables Table structure
 which may perform worse but allow more flexible operations
 like searching / selecting subsets of the data.
 - If None, pd.get_option('io.hdf.default_format') is checked,
 followed by fallback to "fixed"
errors : str, default 'strict'
 Specifies how encoding and decoding errors are to be handled.
 See the errors argument for :func:`open` for a full list
 of options.
encoding : str, default "UTF-8"
min_itemsize : dict or int, optional
 Map column names to minimum string sizes for columns.
nan_rep : Any, optional
 How to represent null values as str.
 Not allowed with append=True.
data_columns : list of columns or True, optional
 List of columns to create as indexed data columns for on-disk
 queries, or True to use all columns. By default only the axes
 of the object are indexed. See :ref:`io.hdf5-query-data-columns`.
 Applicable only to format='table'.
```

## See Also

-----

```
DataFrame.read_hdf : Read from HDF file.
DataFrame.to_parquet : Write a DataFrame to the binary parquet format.
DataFrame.to_sql : Write to a sql table.
DataFrame.to_feather : Write out feather-format for DataFrames.
DataFrame.to_csv : Write out to a csv file.
```

## Examples

-----

```
>>> df = pd.DataFrame({'A': [1, 2, 3], 'B': [4, 5, 6]},
... index=['a', 'b', 'c'])
>>> df.to_hdf('data.h5', key='df', mode='w')
```

We can add another object to the same file:

```
>>> s = pd.Series([1, 2, 3, 4])
>>> s.to_hdf('data.h5', key='s')
```

```
Reading from HDF file:
```

```
>>> pd.read_hdf('data.h5', 'df')
A B
a 1 4
b 2 5
c 3 6
>>> pd.read_hdf('data.h5', 's')
0 1
1 2
2 3
3 4
dtype: int64
```

```
Deleting file with data:
```

```
>>> import os
>>> os.remove('data.h5')
"""
from pandas.io import pytables

pytables.to_hdf(
 path_or_buf,
 key,
 self,
 mode=mode,
 complevel=complevel,
 complib=complib,
 append=append,
 format=format,
 index=index,
 min_itemsize=min_itemsize,
 nan_rep=nan_rep,
 dropna=dropna,
 data_columns=data_columns,
 errors=errors,
 encoding=encoding,
)
```

```
def to_sql(
 self,
 name: str,
 con,
 schema=None,
 if_exists: str = "fail",
 index: bool_t = True,
 index_label=None,
 chunksize=None,
 dtype=None,
 method=None,
) -> None:
"""
Write records stored in a DataFrame to a SQL database.
```

Databases supported by SQLAlchemy [1]\_ are supported. Tables can be newly created, appended to, or overwritten.

Parameters

-----

name : str

Name of SQL table.

con : sqlalchemy.engine.Engine or sqlite3.Connection

Using SQLAlchemy makes it possible to use any DB supported by that

library. Legacy support is provided for `sqlite3.Connection` objects. The user is responsible for engine disposal and connection closure for the SQLAlchemy connectable See `here` [`here`](https://docs.sqlalchemy.org/en/13/core/connections.html).

`schema` : str, optional  
Specify the schema (if database flavor supports this). If `None`, use default schema.

`if_exists` : {'fail', 'replace', 'append'}, default 'fail'  
How to behave if the table already exists.

- \* `fail`: Raise a `ValueError`.
- \* `replace`: Drop the table before inserting new values.
- \* `append`: Insert new values to the existing table.

`index` : bool, default `True`  
Write DataFrame index as a column. Uses `'index_label'` as the column name in the table.

`index_label` : str or sequence, default `None`  
Column label for index column(s). If `None` is given (default) and `'index'` is `True`, then the index names are used.  
A sequence should be given if the DataFrame uses MultiIndex.

`chunksize` : int, optional  
Specify the number of rows in each batch to be written at a time.  
By default, all rows will be written at once.

`dtype` : dict or scalar, optional  
Specifying the datatype for columns. If a dictionary is used, the keys should be the column names and the values should be the SQLAlchemy types or strings for the `sqlite3` legacy mode. If a scalar is provided, it will be applied to all columns.

`method` : {`None`, 'multi', callable}, optional  
Controls the SQL insertion clause used:

- \* `None` : Uses standard SQL ``INSERT`` clause (one per row).
- \* `'multi'`: Pass multiple values in a single ``INSERT`` clause.
- \* `callable` with signature ``(`pd_table`, `conn`, `keys`, `data_iter`)``.

Details and a sample callable implementation can be found in the section :ref:`insert method <io.sql.method>`.

.. versionadded:: 0.24.0

Raises

-----

`ValueError`

When the table already exists and `'if_exists'` is 'fail' (the default).

See Also

-----

`read_sql` : Read a DataFrame from a table.

Notes

-----

Timezone aware datetime columns will be written as ``Timestamp with timezone`` type with SQLAlchemy if supported by the database. Otherwise, the datetimes will be stored as timezone unaware timestamps local to the original timezone.

.. versionadded:: 0.24.0

References

-----

```

.. [1] https://docs.sqlalchemy.org
.. [2] https://www.python.org/dev/peps/pep-0249/

Examples

Create an in-memory SQLite database.

>>> from sqlalchemy import create_engine
>>> engine = create_engine('sqlite://', echo=False)

Create a table from scratch with 3 rows.

>>> df = pd.DataFrame({'name' : ['User 1', 'User 2', 'User 3']})
>>> df
 name
0 User 1
1 User 2
2 User 3

>>> df.to_sql('users', con=engine)
>>> engine.execute("SELECT * FROM users").fetchall()
[(0, 'User 1'), (1, 'User 2'), (2, 'User 3')]

>>> df1 = pd.DataFrame({'name' : ['User 4', 'User 5']})
>>> df1.to_sql('users', con=engine, if_exists='append')
>>> engine.execute("SELECT * FROM users").fetchall()
[(0, 'User 1'), (1, 'User 2'), (2, 'User 3'),
 (0, 'User 4'), (1, 'User 5')]

Overwrite the table with just ``df1``.

>>> df1.to_sql('users', con=engine, if_exists='replace',
... index_label='id')
>>> engine.execute("SELECT * FROM users").fetchall()
[(0, 'User 4'), (1, 'User 5')]

Specify the dtype (especially useful for integers with missing values).
Notice that while pandas is forced to store the data as floating point,
the database supports nullable integers. When fetching the data with
Python, we get back integer scalars.

>>> df = pd.DataFrame({"A": [1, None, 2]})
>>> df
 A
0 1.0
1 NaN
2 2.0

>>> from sqlalchemy.types import Integer
>>> df.to_sql('integers', con=engine, index=False,
... dtype={"A": Integer()})

>>> engine.execute("SELECT * FROM integers").fetchall()
[(1,), (None,), (2,)]
"""

from pandas.io import sql

sql.to_sql(
 self,
 name,
 con,
 schema=schema,
 if_exists=if_exists,

```

```
 index=index,
 index_label=index_label,
 chunksize=chunksize,
 dtype=dtype,
 method=method,
)

def to_pickle(
 self,
 path,
 compression: Optional[str] = "infer",
 protocol: int = pickle.HIGHEST_PROTOCOL,
) -> None:
 """
 Pickle (serialize) object to file.

 Parameters

 path : str
 File path where the pickled object will be stored.
 compression : {'infer', 'gzip', 'bz2', 'zip', 'xz', None}, \
 default 'infer'
 A string representing the compression to use in the output file. By
 default, infers from the file extension in specified path.
 protocol : int
 Int which indicates which protocol should be used by the pickler,
 default HIGHEST_PROTOCOL (see [1]_ paragraph 12.1.2). The possible
 values are 0, 1, 2, 3, 4. A negative value for the protocol
 parameter is equivalent to setting its value to HIGHEST_PROTOCOL.

 .. [1] https://docs.python.org/3/library/pickle.html.

 See Also

 read_pickle : Load pickled pandas object (or any object) from file.
 DataFrame.to_hdf : Write DataFrame to an HDF5 file.
 DataFrame.to_sql : Write DataFrame to a SQL database.
 DataFrame.to_parquet : Write a DataFrame to the binary parquet format.

 Examples

 >>> original_df = pd.DataFrame({"foo": range(5), "bar": range(5, 10)})
 >>> original_df
 foo bar
 0 0 5
 1 1 6
 2 2 7
 3 3 8
 4 4 9
 >>> original_df.to_pickle("./dummy.pkl")

 >>> unpickled_df = pd.read_pickle("./dummy.pkl")
 >>> unpickled_df
 foo bar
 0 0 5
 1 1 6
 2 2 7
 3 3 8
 4 4 9

 >>> import os
 >>> os.remove("./dummy.pkl")
 """
```

```
from pandas.io.pickle import to_pickle

to_pickle(self, path, compression=compression, protocol=protocol)

def to_clipboard(
 self, excel: bool_t = True, sep: Optional[str] = None, **kwargs
) -> None:
 r"""
 Copy object to the system clipboard.

 Write a text representation of object to the system clipboard.
 This can be pasted into Excel, for example.

 Parameters

 excel : bool, default True
 Produce output in a csv format for easy pasting into excel.

 - True, use the provided separator for csv pasting.
 - False, write a string representation of the object to the clipboard.

 sep : str, default ``'\t'``
 Field delimiter.

 **kwargs
 These parameters will be passed to DataFrame.to_csv.

 See Also

 DataFrame.to_csv : Write a DataFrame to a comma-separated values
 (csv) file.
 read_clipboard : Read text from clipboard and pass to read_table.

 Notes

 Requirements for your platform.

 - Linux : `xclip`, or `xsel` (with `PyQt4` modules)
 - Windows : none
 - OS X : none

 Examples

 Copy the contents of a DataFrame to the clipboard.

 >>> df = pd.DataFrame([[1, 2, 3], [4, 5, 6]], columns=['A', 'B', 'C'])

 >>> df.to_clipboard(sep=',') # doctest: +SKIP
 ... # Wrote the following to the system clipboard:
 ... # ,A,B,C
 ... # 0,1,2,3
 ... # 1,4,5,6

 We can omit the index by passing the keyword `index` and setting
 it to false.

 >>> df.to_clipboard(sep=',', index=False) # doctest: +SKIP
 ... # Wrote the following to the system clipboard:
 ... # A,B,C
 ... # 1,2,3
 ... # 4,5,6
 """

 from pandas.io import clipboards
```

```
clipboards.to_clipboard(self, excel=excel, sep=sep, **kwargs)

def to_xarray(self):
 """
 Return an xarray object from the pandas object.

 Returns

 xarray.DataArray or xarray.Dataset
 Data in the pandas structure converted to Dataset if the object is
 a DataFrame, or a DataArray if the object is a Series.

 See Also

 DataFrame.to_hdf : Write DataFrame to an HDF5 file.
 DataFrame.to_parquet : Write a DataFrame to the binary parquet format.

 Notes

 See the `xarray docs <https://xarray.pydata.org/en/stable/>`__

 Examples

 >>> df = pd.DataFrame([('falcon', 'bird', 389.0, 2),
... ('parrot', 'bird', 24.0, 2),
... ('lion', 'mammal', 80.5, 4),
... ('monkey', 'mammal', np.nan, 4)],
... columns=['name', 'class', 'max_speed',
... 'num_legs'])
 >>> df
 name class max_speed num_legs
 0 falcon bird 389.0 2
 1 parrot bird 24.0 2
 2 lion mammal 80.5 4
 3 monkey mammal NaN 4

 >>> df.to_xarray()
<xarray.Dataset>
Dimensions: (index: 4)
Coordinates:
 * index (index) int64 0 1 2 3
Data variables:
 name (index) object 'falcon' 'parrot' 'lion' 'monkey'
 class (index) object 'bird' 'bird' 'mammal' 'mammal'
 max_speed (index) float64 389.0 24.0 80.5 nan
 num_legs (index) int64 2 2 4 4

 >>> df['max_speed'].to_xarray()
<xarray.DataArray 'max_speed' (index: 4)>
array([389., 24., 80.5, nan])
Coordinates:
 * index (index) int64 0 1 2 3

 >>> dates = pd.to_datetime(['2018-01-01', '2018-01-01',
... '2018-01-02', '2018-01-02'])
 >>> df_multiindex = pd.DataFrame({'date': dates,
... 'animal': ['falcon', 'parrot',
... 'falcon', 'parrot'],
... 'speed': [350, 18, 361, 15]})
 >>> df_multiindex = df_multiindex.set_index(['date', 'animal'])

 >>> df_multiindex
 speed
```

```

date animal
2018-01-01 falcon 350
 parrot 18
2018-01-02 falcon 361
 parrot 15

>>> df_multiindex.to_xarray()
<xarray.Dataset>
Dimensions: (animal: 2, date: 2)
Coordinates:
 * date (date) datetime64[ns] 2018-01-01 2018-01-02
 * animal (animal) object 'falcon' 'parrot'
Data variables:
 speed (date, animal) int64 350 18 361 15
"""

xarray = import_optional_dependency("xarray")

if self.ndim == 1:
 return xarray.DataArray.from_series(self)
else:
 return xarray.Dataset.from_dataframe(self)

@Substitution(returns=fmt.return_docstring)
def to_latex(
 self,
 buf=None,
 columns=None,
 col_space=None,
 header=True,
 index=True,
 na_rep="NaN",
 formatters=None,
 float_format=None,
 sparsify=None,
 index_names=True,
 bold_rows=False,
 column_format=None,
 longtable=None,
 escape=None,
 encoding=None,
 decimal=".",
 multicolumn=None,
 multicolumn_format=None,
 multirow=None,
 caption=None,
 label=None,
):
 r"""
 Render object to a LaTeX tabular, longtable, or nested table/tabular.

 Requires ``\usepackage{booktabs}``. The output can be copy/pasted
 into a main LaTeX document or read from an external file
 with ``\input{table.tex}``.

 .. versionchanged:: 0.20.2
 Added to Series.

 .. versionchanged:: 1.0.0
 Added caption and label arguments.

 Parameters

 buf : str, Path or StringIO-like, optional, default None

```

```
 Buffer to write to. If None, the output is returned as a string.
columns : list of label, optional
 The subset of columns to write. Writes all columns by default.
col_space : int, optional
 The minimum width of each column.
header : bool or list of str, default True
 Write out the column names. If a list of strings is given,
 it is assumed to be aliases for the column names.
index : bool, default True
 Write row names (index).
na_rep : str, default 'NaN'
 Missing data representation.
formatters : list of functions or dict of {str: function}, optional
 Formatter functions to apply to columns' elements by position or
 name. The result of each function must be a unicode string.
 List must be of length equal to the number of columns.
float_format : one-parameter function or str, optional, default None
 Formatter for floating point numbers. For example
 ``float_format="%%.2f"`` and ``float_format="{:0.2f}".format`` will
 both result in 0.1234 being formatted as 0.12.
sparsify : bool, optional
 Set to False for a DataFrame with a hierarchical index to print
 every multiindex key at each row. By default, the value will be
 read from the config module.
index_names : bool, default True
 Prints the names of the indexes.
bold_rows : bool, default False
 Make the row labels bold in the output.
column_format : str, optional
 The columns format as specified in `LaTeX table format
 <https://en.wikibooks.org/wiki/LaTeX/Tables>`__ e.g. 'rcl' for 3
 columns. By default, 'l' will be used for all columns except
 columns of numbers, which default to 'r'.
longtable : bool, optional
 By default, the value will be read from the pandas config
 module. Use a longtable environment instead of tabular. Requires
 adding a \usepackage{longtable} to your LaTeX preamble.
escape : bool, optional
 By default, the value will be read from the pandas config
 module. When set to False prevents from escaping latex special
 characters in column names.
encoding : str, optional
 A string representing the encoding to use in the output file,
 defaults to 'utf-8'.
decimal : str, default '.'
 Character recognized as decimal separator, e.g. ',' in Europe.
multicolumn : bool, default True
 Use \multicolumn to enhance MultiIndex columns.
 The default will be read from the config module.
multicolumn_format : str, default 'l'
 The alignment for multicolumns, similar to `column_format`_
 The default will be read from the config module.
multirow : bool, default False
 Use \multirow to enhance MultiIndex rows. Requires adding a
 \usepackage{multirow} to your LaTeX preamble. Will print
 centered labels (instead of top-aligned) across the contained
 rows, separating groups via clines. The default will be read
 from the pandas config module.
caption : str, optional
 The LaTeX caption to be placed inside ``\caption{}`` in the output.
.. versionadded:: 1.0.0
```

```
label : str, optional
 The LaTeX label to be placed inside ``\label{}`` in the output.
 This is used with ``\ref{}`` in the main ``.tex`` file.

 .. versionadded:: 1.0.0
%(returns)s
See Also

DataFrame.to_string : Render a DataFrame to a console-friendly
 tabular output.
DataFrame.to_html : Render a DataFrame as an HTML table.

Examples

>>> df = pd.DataFrame({'name': ['Raphael', 'Donatello'],
... 'mask': ['red', 'purple'],
... 'weapon': ['sai', 'bo staff']})
>>> print(df.to_latex(index=False)) # doctest: +NORMALIZE_WHITESPACE
\begin{tabular}{lll}
\toprule
 name & mask & weapon \\
\midrule
 Raphael & red & sai \\
 Donatello & purple & bo staff \\
\bottomrule
\end{tabular}
"""
Get defaults from the pandas config
if self.ndim == 1:
 self = self.to_frame()
if longtable is None:
 longtable = config.get_option("display.latex.longtable")
if escape is None:
 escape = config.get_option("display.latex.escape")
if multicolumn is None:
 multicolumn = config.get_option("display.latex.multicolumn")
if multicolumn_format is None:
 multicolumn_format = config.get_option("display.latex.multicolumn_format")
if multirow is None:
 multirow = config.get_option("display.latex.multirow")

formatter = DataFrameFormatter(
 self,
 columns=columns,
 col_space=col_space,
 na_rep=na_rep,
 header=header,
 index=index,
 formatters=formatters,
 float_format=float_format,
 bold_rows=bold_rows,
 sparsify=sparsify,
 index_names=index_names,
 escape=escape,
 decimal=decimal,
)
return formatter.to_latex(
 buf=buf,
 column_format=column_format,
 longtable=longtable,
 encoding=encoding,
 multicolumn=multicolumn,
 multicolumn_format=multicolumn_format,
```

```
 multirow=multirow,
 caption=caption,
 label=label,
)

def to_csv(
 self,
 path_or_buf: Optional[FilePathOrBuffer] = None,
 sep: str = ",",
 na_rep: str = "",
 float_format: Optional[str] = None,
 columns: Optional[Sequence[Label]] = None,
 header: Union[bool_t, List[str]] = True,
 index: bool_t = True,
 index_label: Optional[Union[bool_t, str, Sequence[Label]]] = None,
 mode: str = "w",
 encoding: Optional[str] = None,
 compression: Optional[Union[str, Mapping[str, str]]] = "infer",
 quoting: Optional[int] = None,
 quotechar: str = "'",
 line_terminator: Optional[str] = None,
 chunksize: Optional[int] = None,
 date_format: Optional[str] = None,
 doublequote: bool_t = True,
 escapechar: Optional[str] = None,
 decimal: Optional[str] = ".",
) -> Optional[str]:
 r"""
 Write object to a comma-separated values (csv) file.

 .. versionchanged:: 0.24.0
 The order of arguments for Series was changed.

 Parameters

 path_or_buf : str or file handle, default None
 File path or object, if None is provided the result is returned as
 a string. If a file object is passed it should be opened with
 `newline=''`, disabling universal newlines.

 .. versionchanged:: 0.24.0

 Was previously named "path" for Series.

 sep : str, default ','
 String of length 1. Field delimiter for the output file.
 na_rep : str, default ''
 Missing data representation.
 float_format : str, default None
 Format string for floating point numbers.
 columns : sequence, optional
 Columns to write.
 header : bool or list of str, default True
 Write out the column names. If a list of strings is given it is
 assumed to be aliases for the column names.

 .. versionchanged:: 0.24.0

 Previously defaulted to False for Series.

 index : bool, default True
 Write row names (index).
 index_label : str or sequence, or False, default None
```

Column label for index column(s) if desired. If None is given, and `header` and `index` are True, then the index names are used. A sequence should be given if the object uses MultiIndex. If False do not print fields for index names. Use index\_label=False for easier importing in R.

**mode** : str  
 Python write mode, default 'w'.

**encoding** : str, optional  
 A string representing the encoding to use in the output file, defaults to 'utf-8'.

**compression** : str or dict, default 'infer'  
 If str, represents compression mode. If dict, value at 'method' is the compression mode. Compression mode may be any of the following possible values: {'infer', 'gzip', 'bz2', 'zip', 'xz', None}. If compression mode is 'infer' and `path\_or\_buf` is path-like, then detect compression mode from the following extensions: '.gz', '.bz2', '.zip' or '.xz'. (otherwise no compression). If dict given and mode is one of {'zip', 'gzip', 'bz2'}, or inferred as one of the above, other entries passed as additional compression options.

.. versionchanged:: 1.0.0  
 May now be a dict with key 'method' as compression mode and other entries as additional compression options if compression mode is 'zip'.

.. versionchanged:: 1.1.0  
 Passing compression options as keys in dict is supported for compression modes 'gzip' and 'bz2' as well as 'zip'.

**quoting** : optional constant from csv module  
 Defaults to csv.QUOTE\_MINIMAL. If you have set a `float\_format` then floats are converted to strings and thus csv.QUOTE\_NONNUMERIC will treat them as non-numeric.

**quotechar** : str, default '\"'  
 String of length 1. Character used to quote fields.

**line\_terminator** : str, optional  
 The newline character or character sequence to use in the output file. Defaults to `os.linesep`, which depends on the OS in which this method is called ('\n' for linux, '\r\n' for Windows, i.e.).

.. versionchanged:: 0.24.0

**chunksize** : int or None  
 Rows to write at a time.

**date\_format** : str, default None  
 Format string for datetime objects.

**doublequote** : bool, default True  
 Control quoting of `quotechar` inside a field.

**escapechar** : str, default None  
 String of length 1. Character used to escape `sep` and `quotechar` when appropriate.

**decimal** : str, default '.'  
 Character recognized as decimal separator. E.g. use ',' for European data.

**Returns**  
 -----  
 None or str  
 If path\_or\_buf is None, returns the resulting csv format as a string. Otherwise returns None.

```

See Also

read_csv : Load a CSV file into a DataFrame.
to_excel : Write DataFrame to an Excel file.

Examples

>>> df = pd.DataFrame({'name': ['Raphael', 'Donatello'],
... 'mask': ['red', 'purple'],
... 'weapon': ['sai', 'bo staff']})
>>> df.to_csv(index=False)
'name,mask,weapon\nRaphael,red,sai\nDonatello,purple,bo staff\n'

Create 'out.zip' containing 'out.csv'

>>> compression_opts = dict(method='zip',
... archive_name='out.csv') # doctest: +SKIP
>>> df.to_csv('out.zip', index=False,
... compression=compression_opts) # doctest: +SKIP
"""
df = self if isinstance(self, ABCDataFrame) else self.to_frame()

from pandas.io.formats.csvs import CSVFormatter

formatter = CSVFormatter(
 df,
 path_or_buf,
 line_terminator=line_terminator,
 sep=sep,
 encoding=encoding,
 compression=compression,
 quoting=quoting,
 na_rep=na_rep,
 float_format=float_format,
 cols=columns,
 header=header,
 index=index,
 index_label=index_label,
 mode=mode,
 chunksize=chunksize,
 quotechar=quotechar,
 date_format=date_format,
 doublequote=doublequote,
 escapechar=escapechar,
 decimal=decimal,
)
formatter.save()

if path_or_buf is None:
 return formatter.path_or_buf.getvalue()

return None

Lookup Caching

def _set_as_cached(self, item, cacher) -> None:
 """
 Set the _cacher attribute on the calling object with a weakref to
 cacher.
 """
 self._cacher = (item, weakref.ref(cacher))

```

```

def _reset_cacher(self) -> None:
 """
 Reset the cacher.
 """
 if hasattr(self, "_cacher"):
 del self._cacher

def _maybe_cache_changed(self, item, value) -> None:
 """
 The object has called back to us saying maybe it has changed.
 """
 loc = self._info_axis.get_loc(item)
 self._mgr.iset(loc, value)

@property
def _is_cached(self) -> bool_t:
 """Return boolean indicating if self is cached or not."""
 return getattr(self, "_cacher", None) is not None

def _get_cacher(self):
 """return my cacher or None"""
 cacher = getattr(self, "_cacher", None)
 if cacher is not None:
 cacher = cacher[1]()
 return cacher

def _maybe_update_cacher(
 self, clear: bool_t = False, verify_is_copy: bool_t = True
) -> None:
 """
 See if we need to update our parent cacher if clear, then clear our
 cache.

 Parameters

 clear : bool, default False
 Clear the item cache.
 verify_is_copy : bool, default True
 Provide is_copy checks.
 """
 cacher = getattr(self, "_cacher", None)
 if cacher is not None:
 ref = cacher[1]()

 # we are trying to reference a dead referant, hence
 # a copy
 if ref is None:
 del self._cacher
 else:
 if len(self) == len(ref):
 # otherwise, either self or ref has swapped in new arrays
 ref._maybe_cache_changed(cacher[0], self)

 if verify_is_copy:
 self._check_setitem_copy(stacklevel=5, t="referant")

 if clear:
 self._clear_item_cache()

def _clear_item_cache(self) -> None:
 self._item_cache.clear()

```

```

Indexing Methods

def take(
 self: FrameOrSeries, indices, axis=0, is_copy: Optional[bool_t] = None, **kwargs
) -> FrameOrSeries:
 """
 Return the elements in the given *positional* indices along an axis.

 This means that we are not indexing according to actual values in
 the index attribute of the object. We are indexing according to the
 actual position of the element in the object.

 Parameters

 indices : array-like
 An array of ints indicating which positions to take.
 axis : {0 or 'index', 1 or 'columns', None}, default 0
 The axis on which to select elements. ``0`` means that we are
 selecting rows, ``1`` means that we are selecting columns.
 is_copy : bool
 Before pandas 1.0, ``is_copy=False`` can be specified to ensure
 that the return value is an actual copy. Starting with pandas 1.0,
 ``take`` always returns a copy, and the keyword is therefore
 deprecated.

 .. deprecated:: 1.0.0
 **kwargs
 For compatibility with :meth:`numpy.take`. Has no effect on the
 output.

 Returns

 taken : same type as caller
 An array-like containing the elements taken from the object.

 See Also

 DataFrame.loc : Select a subset of a DataFrame by labels.
 DataFrame.iloc : Select a subset of a DataFrame by positions.
 numpy.take : Take elements from an array along an axis.

 Examples

 >>> df = pd.DataFrame([('falcon', 'bird', 389.0),
... ('parrot', 'bird', 24.0),
... ('lion', 'mammal', 80.5),
... ('monkey', 'mammal', np.nan)],
... columns=['name', 'class', 'max_speed'],
... index=[0, 2, 3, 1])
 >>> df
 name class max_speed
 0 falcon bird 389.0
 2 parrot bird 24.0
 3 lion mammal 80.5
 1 monkey mammal NaN

 Take elements at positions 0 and 3 along the axis 0 (default).

 Note how the actual indices selected (0 and 1) do not correspond to
 our selected indices 0 and 3. That's because we are selecting the 0th
 and 3rd rows, not rows whose indices equal 0 and 3.

```

```
>>> df.take([0, 3])
 name class max_speed
0 falcon bird 389.0
1 monkey mammal NaN
```

Take elements at indices 1 and 2 along the axis 1 (column selection).

```
>>> df.take([1, 2], axis=1)
 class max_speed
0 bird 389.0
2 bird 24.0
3 mammal 80.5
1 mammal NaN
```

We may take elements using negative integers for positive indices, starting from the end of the object, just like with Python lists.

```
>>> df.take([-1, -2])
 name class max_speed
1 monkey mammal NaN
3 lion mammal 80.5
"""
if is_copy is not None:
 warnings.warn(
 "is_copy is deprecated and will be removed in a future version. "
 "'take' always returns a copy, so there is no need to specify this.",
 FutureWarning,
 stacklevel=2,
)

```

```
nv.validate_take(tuple(), kwargs)
```

```
self._consolidate_inplace()
```

```
new_data = self._mgr.take(
 indices, axis=self._get_block_manager_axis(axis), verify=True
)
return self._constructor(new_data).__finalize__(self, method="take")
```

```
def _take_with_is_copy(self: FrameOrSeries, indices, axis=0) -> FrameOrSeries:
 """
 Internal version of the `take` method that sets the `_is_copy` attribute to keep track of the parent dataframe (using in indexing for the SettingWithCopyWarning).

```

See the docstring of `take` for full explanation of the parameters.

```
 result = self.take(indices=indices, axis=axis)
 # Maybe set copy if we didn't actually change the index.
 if not result._get_axis(axis).equals(self._get_axis(axis)):
 result._set_is_copy(self)
 return result
```

```
def xs(self, key, axis=0, level=None, drop_level: bool_t = True):
 """
 Return cross-section from the Series/DataFrame.
```

This method takes a `key` argument to select data at a particular level of a MultiIndex.

Parameters

-----

key : label or tuple of label

Label contained in the index, or partially in a MultiIndex.  
axis : {0 or 'index', 1 or 'columns'}, default 0  
    Axis to retrieve cross-section on.  
level : object, defaults to first n levels (n=1 or len(key))  
    In case of a key partially contained in a MultiIndex, indicate  
    which levels are used. Levels can be referred by label or position.  
drop\_level : bool, default True  
    If False, returns object with same levels as self.

Returns

-----

Series or DataFrame

Cross-section from the original Series or DataFrame  
corresponding to the selected index levels.

See Also

-----

DataFrame.loc : Access a group of rows and columns  
    by label(s) or a boolean array.

DataFrame.iloc : Purely integer-location based indexing  
    for selection by position.

Notes

-----

`xs` can not be used to set values.

MultiIndex Slicers is a generic way to get/set values on  
any level or levels.

It is a superset of `xs` functionality, see  
[:ref:`MultiIndex Slicers <advanced.mi\\_slicers>](#).

Examples

-----

```
>>> d = {'num_legs': [4, 4, 2, 2],
... 'num_wings': [0, 0, 2, 2],
... 'class': ['mammal', 'mammal', 'mammal', 'bird'],
... 'animal': ['cat', 'dog', 'bat', 'penguin'],
... 'locomotion': ['walks', 'walks', 'flies', 'walks']}
>>> df = pd.DataFrame(data=d)
>>> df = df.set_index(['class', 'animal', 'locomotion'])
>>> df
```

|       |        |         | num_legs | num_wings |   |
|-------|--------|---------|----------|-----------|---|
| class | mammal | cat     | walks    | 4         | 0 |
|       |        | dog     | walks    | 4         | 0 |
|       |        | bat     | flies    | 2         | 2 |
|       | bird   | penguin | walks    | 2         | 2 |

Get values at specified index

```
>>> df.xs('mammal')
 num_legs num_wings
animal locomotion
cat walks 4 0
dog walks 4 0
bat flies 2 2
```

Get values at several indexes

```
>>> df.xs([('mammal', 'dog'))
 num_legs num_wings
locomotion
walks 4 0
```

```

Get values at specified index and level

>>> df.xs('cat', level=1)
 num_legs num_wings
class locomotion
mammal walks 4 0

Get values at several indexes and levels

>>> df.xs([('bird', 'walks'),
... level=[0, 'locomotion'])
 num_legs num_wings
animal
penguin 2 2

Get values at specified column and axis

>>> df.xs('num_wings', axis=1)
class animal locomotion
mammal cat walks 0
 dog walks 0
 bat flies 2
bird penguin walks 2
Name: num_wings, dtype: int64
"""

axis = self._get_axis_number(axis)
labels = self._get_axis(axis)
if level is not None:
 if not isinstance(labels, MultiIndex):
 raise TypeError("Index must be a MultiIndex")
 loc, new_ax = labels.get_loc_level(key, level=level, drop_level=drop_level)

 # create the tuple of the indexer
 _indexer = [slice(None)] * self.ndim
 _indexer[axis] = loc
 indexer = tuple(_indexer)

 result = self.iloc[indexer]
 setattr(result, result._get_axis_name(axis), new_ax)
 return result

if axis == 1:
 return self[key]

self._consolidate_inplace()

index = self.index
if isinstance(index, MultiIndex):
 loc, new_index = self.index.get_loc_level(key, drop_level=drop_level)
else:
 loc = self.index.get_loc(key)

 if isinstance(loc, np.ndarray):
 if loc.dtype == np.bool_:
 (inds,) = loc.nonzero()
 return self._take_with_is_copy(inds, axis=axis)
 else:
 return self._take_with_is_copy(loc, axis=axis)

 if not is_scalar(loc):
 new_index = self.index[loc]

```

```

if is_scalar(loc):
 # In this case loc should be an integer
 if self.ndim == 1:
 # if we encounter an array-like and we only have 1 dim
 # that means that their are list/ndarrays inside the Series!
 # so just return them (GH 6394)
 return self._values[loc]

 new_values = self._mgr.fast_xs(loc)

 result = self._constructor_sliced(
 new_values,
 index=self.columns,
 name=self.index[loc],
 dtype=new_values.dtype,
)

else:
 result = self.iloc[loc]
 result.index = new_index

this could be a view
but only in a single-dtyped view sliceable case
result._set_is_copy(self, copy=not result._is_view)
return result

def __getitem__(self, item):
 raise AbstractMethodError(self)

def _get_item_cache(self, item):
 """Return the cached item, item represents a label indexer."""
 cache = self._item_cache
 res = cache.get(item)
 if res is None:
 # All places that call _get_item_cache have unique columns,
 # pending resolution of GH#33047

 loc = self.columns.get_loc(item)
 values = self._mgr.iget(loc)
 res = self._box_col_values(values, loc)

 cache[item] = res
 res._set_as_cached(item, self)

 # for a chain
 res._is_copy = self._is_copy
 return res

def _slice(self: FrameOrSeries, slobj: slice, axis=0) -> FrameOrSeries:
 """
 Construct a slice of this container.

 Slicing with this method is *always* positional.
 """
 assert isinstance(slobj, slice), type(slobj)
 axis = self._get_block_manager_axis(axis)
 result = self._constructor(self._mgr.get_slice(slobj, axis=axis))
 result = result.__finalize__(self)

 # this could be a view
 # but only in a single-dtyped view sliceable case
 is_copy = axis != 0 or result._is_view
 result._set_is_copy(self, copy=is_copy)

```

```

 return result

def _iset_item(self, loc: int, value) -> None:
 self._mgr.iset(loc, value)
 self._clear_item_cache()

def _set_item(self, key, value) -> None:
 try:
 loc = self._info_axis.get_loc(key)
 except KeyError:
 # This item wasn't present, just insert at end
 self._mgr.insert(len(self._info_axis), key, value)
 return

NDFrame._iset_item(self, loc, value)

def _set_is_copy(self, ref, copy: bool_t = True) -> None:
 if not copy:
 self._is_copy = None
 else:
 assert ref is not None
 self._is_copy = weakref.ref(ref)

def _check_is_chained_assignment_possible(self) -> bool_t:
 """
 Check if we are a view, have a cacher, and are of mixed type.
 If so, then force a setitem_copy check.

 Should be called just near setting a value

 Will return a boolean if it we are a view and are cached, but a
 single-dtype meaning that the cacher should be updated following
 setting.
 """
 if self._is_view and self._is_cached:
 ref = self._get_cacher()
 if ref is not None and ref._is_mixed_type:
 self._check_setitem_copy(stacklevel=4, t="referant", force=True)
 return True
 elif self._is_copy:
 self._check_setitem_copy(stacklevel=4, t="referant")
 return False

def _check_setitem_copy(self, stacklevel=4, t="setting", force=False):
 """
 Parameters

 stacklevel : int, default 4
 the level to show of the stack when the error is output
 t : str, the type of setting error
 force : bool, default False
 If True, then force showing an error.

 validate if we are doing a setitem on a chained copy.

 If you call this function, be sure to set the stacklevel such that the
 user will see the error *at the level of setting*

 It is technically possible to figure out that we are setting on
 a copy even WITH a multi-dtyped pandas object. In other words, some
 blocks may be views while other are not. Currently _is_view will ALWAYS
 return False for multi-blocks to avoid having to handle this case.
 """

```

```

df = DataFrame(np.arange(0, 9), columns=['count'])
df['group'] = 'b'

This technically need not raise SettingWithCopy if both are view
(which is not # generally guaranteed but is usually True. However,
this is in general not a good practice and we recommend using .loc.
df.iloc[0:5]['group'] = 'a'

"""
return early if the check is not needed
if not (force or self._is_copy):
 return

value = config.get_option("mode.chained_assignment")
if value is None:
 return

see if the copy is not actually referred; if so, then dissolve
the copy weakref
if self._is_copy is not None and not isinstance(self._is_copy, str):
 r = self._is_copy()
 if not gc.get_referents(r) or r.shape == self.shape:
 self._is_copy = None
 return

a custom message
if isinstance(self._is_copy, str):
 t = self._is_copy

elif t == "referant":
 t = (
 "\n"
 "A value is trying to be set on a copy of a slice from a "
 "DataFrame\n\n"
 "See the caveats in the documentation: "
 "https://pandas.pydata.org/pandas-docs/stable/user_guide/"
 "indexing.html#returning-a-view-versus-a-copy"
)

else:
 t = (
 "\n"
 "A value is trying to be set on a copy of a slice from a "
 "DataFrame.\n"
 "Try using .loc[row_indexer,col_indexer] = value "
 "instead\n\nSee the caveats in the documentation: "
 "https://pandas.pydata.org/pandas-docs/stable/user_guide/"
 "indexing.html#returning-a-view-versus-a-copy"
)

if value == "raise":
 raise com.SettingWithCopyError(t)
elif value == "warn":
 warnings.warn(t, com.SettingWithCopyWarning, stacklevel=stacklevel)

def __delitem__(self, key) -> None:
 """
 Delete item
 """
 deleted = False

 maybe_shortcut = False

```

```

if self.ndim == 2 and isinstance(self.columns, MultiIndex):
 try:
 maybe_shortcut = key not in self.columns._engine
 except TypeError:
 pass

if maybe_shortcut:
 # Allow shorthand to delete all columns whose first len(key)
 # elements match key:
 if not isinstance(key, tuple):
 key = (key,)
 for col in self.columns:
 if isinstance(col, tuple) and col[: len(key)] == key:
 del self[col]
 deleted = True
 if not deleted:
 # If the above loop ran and didn't delete anything because
 # there was no match, this call should raise the appropriate
 # exception:
 loc = self.axes[-1].get_loc(key)
 self._mgr.idelete(loc)

 # delete from the caches
 try:
 del self._item_cache[key]
 except KeyError:
 pass

Unsorted

def get(self, key, default=None):
 """
 Get item from object for given key (ex: DataFrame column).

 Returns default value if not found.

 Parameters

 key : object

 Returns

 value : same type as items contained in object
 """
 try:
 return self[key]
 except (KeyError, ValueError, IndexError):
 return default

@property
def _is_view(self) -> bool_t:
 """Return boolean indicating if self is view of another array"""
 return self._mgr.is_view

def reindex_like(
 self: FrameOrSeries,
 other,
 method: Optional[str] = None,
 copy: bool_t = True,
 limit=None,
 tolerance=None,
) -> FrameOrSeries:

```

```
"""
Return an object with matching indices as other object.

Conform the object to the same index on all axes. Optional
filling logic, placing NaN in locations having no value
in the previous index. A new object is produced unless the
new index is equivalent to the current one and copy=False.

Parameters

other : Object of the same data type
 Its row and column indices are used to define the new indices
 of this object.
method : {None, 'backfill'/'bfill', 'pad'/'ffill', 'nearest'}
 Method to use for filling holes in reindexed DataFrame.
 Please note: this is only applicable to DataFrames/Series with a
 monotonically increasing/decreasing index.

 * None (default): don't fill gaps
 * pad / ffill: propagate last valid observation forward to next
 valid
 * backfill / bfill: use next valid observation to fill gap
 * nearest: use nearest valid observations to fill gap.

copy : bool, default True
 Return a new object, even if the passed indexes are the same.
limit : int, default None
 Maximum number of consecutive labels to fill for inexact matches.
tolerance : optional
 Maximum distance between original and new labels for inexact
 matches. The values of the index at the matching locations must
 satisfy the equation ``abs(index[indexer] - target) <= tolerance``.

 Tolerance may be a scalar value, which applies the same tolerance
 to all values, or list-like, which applies variable tolerance per
 element. List-like includes list, tuple, array, Series, and must be
 the same size as the index and its dtype must exactly match the
 index's type.

Returns

Series or DataFrame
 Same type as caller, but with changed indices on each axis.

See Also

DataFrame.set_index : Set row labels.
DataFrame.reset_index : Remove row labels or move them to new columns.
DataFrame.reindex : Change to new indices or expand indices.

Notes

Same as calling
```.reindex(index=other.index, columns=other.columns,...)```.

Examples
-----
>>> df1 = pd.DataFrame([[24.3, 75.7, 'high'],
...                      [31, 87.8, 'high'],
...                      [22, 71.6, 'medium'],
...                      [35, 95, 'medium']],
...                     columns=['temp_celsius', 'temp_fahrenheit',
...                               'windspeed'],
```

```

...                               index=pd.date_range(start='2014-02-12',
...                                         end='2014-02-15', freq='D'))
...
>>> df1
      temp_celsius  temp_fahrenheit  windspeed
2014-02-12        24.3            75.7    high
2014-02-13        31.0            87.8    high
2014-02-14        22.0            71.6  medium
2014-02-15        35.0            95.0  medium

>>> df2 = pd.DataFrame([[28, 'low'],
...                      [30, 'low'],
...                      [35.1, 'medium']],
...                     columns=['temp_celsius', 'windspeed'],
...                     index=pd.DatetimeIndex(['2014-02-12', '2014-02-13',
...                                              '2014-02-15']))
...
>>> df2
      temp_celsius  windspeed
2014-02-12        28.0      low
2014-02-13        30.0      low
2014-02-15        35.1  medium

>>> df2.reindex_like(df1)
      temp_celsius  temp_fahrenheit  windspeed
2014-02-12        28.0            NaN      low
2014-02-13        30.0            NaN      low
2014-02-14        NaN            NaN      NaN
2014-02-15        35.1            NaN  medium
"""
d = other._construct_axes_dict(
    axes=self._AXIS_ORDERS,
    method=method,
    copy=copy,
    limit=limit,
    tolerance=tolerance,
)
return self.reindex(**d)

def drop(
    self,
    labels=None,
    axis=0,
    index=None,
    columns=None,
    level=None,
    inplace: bool_t = False,
    errors: str = "raise",
):
    """
    inplace = validate_bool_kwarg(inplace, "inplace")

    if labels is not None:
        if index is not None or columns is not None:
            raise ValueError("Cannot specify both 'labels' and 'index'/'columns'")
        axis_name = self._get_axis_name(axis)
        axes = {axis_name: labels}
    elif index is not None or columns is not None:
        axes, _ = self._construct_axes_from_arguments((index, columns), {})
    else:
        raise ValueError(
            "Need to specify at least one of 'labels', 'index' or 'columns'"
        )

```

```

        )

obj = self

for axis, labels in axes.items():
    if labels is not None:
        obj = obj._drop_axis(labels, axis, level=level, errors=errors)

if inplace:
    self._update_inplace(obj)
else:
    return obj

def _drop_axis(
    self: FrameOrSeries, labels, axis, level=None, errors: str = "raise"
) -> FrameOrSeries:
    """
    Drop labels from specified axis. Used in the ``drop`` method
    internally.

    Parameters
    -----
    labels : single label or list-like
    axis : int or axis name
    level : int or level name, default None
        For MultiIndex
    errors : {'ignore', 'raise'}, default 'raise'
        If 'ignore', suppress error and existing labels are dropped.

    """
    axis = self._get_axis_number(axis)
    axis_name = self._get_axis_name(axis)
    axis = self._get_axis(axis)

    if axis.is_unique:
        if level is not None:
            if not isinstance(axis, MultiIndex):
                raise AssertionError("axis must be a MultiIndex")
            new_axis = axis.drop(labels, level=level, errors=errors)
        else:
            new_axis = axis.drop(labels, errors=errors)
        result = self.reindex(**{axis_name: new_axis})
    else:
        labels = ensure_object(com.index_labels_to_array(labels))
        if level is not None:
            if not isinstance(axis, MultiIndex):
                raise AssertionError("axis must be a MultiIndex")
            indexer = ~axis.get_level_values(level).isin(labels)

            # GH 18561 MultiIndex.drop should raise if label is absent
            if errors == "raise" and indexer.all():
                raise KeyError(f"{labels} not found in axis")
        else:
            indexer = ~axis.isin(labels)
            # Check if label doesn't exist along axis
            labels_missing = (axis.get_indexer_for(labels) == -1).any()
            if errors == "raise" and labels_missing:
                raise KeyError(f"{labels} not found in axis")

    slicer = [slice(None)] * self.ndim
    slicer[self._get_axis_number(axis_name)] = indexer

```

```
        result = self.loc[tuple(slicer)]  
  
    return result  
  
def _update_inplace(self, result, verify_is_copy: bool_t = True) -> None:  
    """  
    Replace self internals with result.  
  
    Parameters  
    -----  
    result : same type as self  
    verify_is_copy : bool, default True  
        Provide is_copy checks.  
    """  
    # NOTE: This does *not* call __finalize__ and that's an explicit  
    # decision that we may revisit in the future.  
    self._reset_cache()  
    self._clear_item_cache()  
    self._mgr = result._mgr  
    self._maybe_update_cacher(verify_is_copy=verify_is_copy)  
  
def add_prefix(self: FrameOrSeries, prefix: str) -> FrameOrSeries:  
    """  
    Prefix labels with string `prefix`.  
  
    For Series, the row labels are prefixed.  
    For DataFrame, the column labels are prefixed.  
  
    Parameters  
    -----  
    prefix : str  
        The string to add before each label.  
  
    Returns  
    -----  
    Series or DataFrame  
        New Series or DataFrame with updated labels.  
  
    See Also  
    -----  
    Series.add_suffix: Suffix row labels with string `suffix`.  
    DataFrame.add_suffix: Suffix column labels with string `suffix`.  
  
    Examples  
    -----  
    >>> s = pd.Series([1, 2, 3, 4])  
    >>> s  
    0    1  
    1    2  
    2    3  
    3    4  
    dtype: int64  
  
    >>> s.add_prefix('item_')  
    item_0    1  
    item_1    2  
    item_2    3  
    item_3    4  
    dtype: int64  
  
    >>> df = pd.DataFrame({'A': [1, 2, 3, 4], 'B': [3, 4, 5, 6]})  
    >>> df
```

```
A   B
0   1   3
1   2   4
2   3   5
3   4   6

>>> df.add_prefix('col_')
      col_A  col_B
0          1      3
1          2      4
2          3      5
3          4      6
"""
f = functools.partial("{prefix}{}".format, prefix=prefix)

mapper = {self._info_axis_name: f}
return self.rename(**mapper) # type: ignore

def add_suffix(self: FrameOrSeries, suffix: str) -> FrameOrSeries:
    """
    Suffix labels with string `suffix`.

    For Series, the row labels are suffixed.
    For DataFrame, the column labels are suffixed.

    Parameters
    -----
    suffix : str
        The string to add after each label.

    Returns
    -----
    Series or DataFrame
        New Series or DataFrame with updated labels.

    See Also
    -----
    Series.add_prefix: Prefix row labels with string `prefix`.
    DataFrame.add_prefix: Prefix column labels with string `prefix`.

    Examples
    -----
    >>> s = pd.Series([1, 2, 3, 4])
    >>> s
    0    1
    1    2
    2    3
    3    4
    dtype: int64

    >>> s.add_suffix('_item')
    0_item    1
    1_item    2
    2_item    3
    3_item    4
    dtype: int64

    >>> df = pd.DataFrame({'A': [1, 2, 3, 4], 'B': [3, 4, 5, 6]})
    >>> df
      A   B
0   1   3
1   2   4
2   3   5
```

```

3 4 6

>>> df.add_suffix('_col')
      A_col  B_col
0           1     3
1           2     4
2           3     5
3           4     6
"""
f = functools.partial("{}{suffix}".format, suffix=suffix)

mapper = {self._info_axis_name: f}
return self.rename(**mapper)  # type: ignore

def sort_values(
    self,
    axis=0,
    ascending=True,
    inplace: bool_t = False,
    kind: str = "quicksort",
    na_position: str = "last",
    ignore_index: bool_t = False,
):
    """
    Sort by the values along either axis.

    Parameters
    -----
    %(optional_by)s
    axis : %(axes_single_arg)s, default 0
        Axis to be sorted.
    ascending : bool or list of bool, default True
        Sort ascending vs. descending. Specify list for multiple sort
        orders. If this is a list of bools, must match the length of
        the by.
    inplace : bool, default False
        If True, perform operation in-place.
    kind : {'quicksort', 'mergesort', 'heapsort'}, default 'quicksort'
        Choice of sorting algorithm. See also ndarray.np.sort for more
        information. `mergesort` is the only stable algorithm. For
        DataFrames, this option is only applied when sorting on a single
        column or label.
    na_position : {'first', 'last'}, default 'last'
        Puts NaNs at the beginning if `first`; `last` puts NaNs at the
        end.
    ignore_index : bool, default False
        If True, the resulting axis will be labeled 0, 1, ..., n - 1.

    .. versionadded:: 1.0.0

    Returns
    -----
    sorted_obj : DataFrame or None
        DataFrame with sorted values if inplace=False, None otherwise.

    Examples
    -----
    >>> df = pd.DataFrame({
    ...     'col1': ['A', 'A', 'B', np.nan, 'D', 'C'],
    ...     'col2': [2, 1, 9, 8, 7, 4],
    ...     'col3': [0, 1, 9, 4, 2, 3],
    ... })
    >>> df
       col1  col2  col3

```

```
0   A   2   0
1   A   1   1
2   B   9   9
3  NaN  8   4
4   D   7   2
5   C   4   3
```

Sort by col1

```
>>> df.sort_values(by=['col1'])
      col1  col2  col3
0     A     2     0
1     A     1     1
2     B     9     9
5     C     4     3
4     D     7     2
3  NaN     8     4
```

Sort by multiple columns

```
>>> df.sort_values(by=['col1', 'col2'])
      col1  col2  col3
1     A     1     1
0     A     2     0
2     B     9     9
5     C     4     3
4     D     7     2
3  NaN     8     4
```

Sort Descending

```
>>> df.sort_values(by='col1', ascending=False)
      col1  col2  col3
4     D     7     2
5     C     4     3
2     B     9     9
0     A     2     0
1     A     1     1
3  NaN     8     4
```

Putting NAs first

```
>>> df.sort_values(by='col1', ascending=False, na_position='first')
      col1  col2  col3
3  NaN     8     4
4     D     7     2
5     C     4     3
2     B     9     9
0     A     2     0
1     A     1     1
"""
raise AbstractMethodError(self)

def reindex(self: FrameOrSeries, *args, **kwargs) -> FrameOrSeries:
    """
    Conform %(klass)s to new index with optional filling logic.
```

Places NA/NaN in locations having no value in the previous index. A new object is produced unless the new index is equivalent to the current one and ``copy=False``.

Parameters

```

%(optional_labels)s
%(axes)s : array-like, optional
    New labels / index to conform to, should be specified using
    keywords. Preferably an Index object to avoid duplicating data.
%(optional_axis)s
method : {None, 'backfill'/'bfill', 'pad'/'ffill', 'nearest'}
    Method to use for filling holes in reindexed DataFrame.
    Please note: this is only applicable to DataFrames/Series with a
    monotonically increasing/decreasing index.

    * None (default): don't fill gaps
    * pad / ffill: Propagate last valid observation forward to next
        valid.
    * backfill / bfill: Use next valid observation to fill gap.
    * nearest: Use nearest valid observations to fill gap.

copy : bool, default True
    Return a new object, even if the passed indexes are the same.
level : int or name
    Broadcast across a level, matching Index values on the
    passed MultiIndex level.
fill_value : scalar, default np.NaN
    Value to use for missing values. Defaults to NaN, but can be any
    "compatible" value.
limit : int, default None
    Maximum number of consecutive elements to forward or backward fill.
tolerance : optional
    Maximum distance between original and new labels for inexact
    matches. The values of the index at the matching locations must
    satisfy the equation ``abs(index[indexer] - target) <= tolerance``.

    Tolerance may be a scalar value, which applies the same tolerance
    to all values, or list-like, which applies variable tolerance per
    element. List-like includes list, tuple, array, Series, and must be
    the same size as the index and its dtype must exactly match the
    index's type.

Returns
-----
%(klass)s with changed index.

See Also
-----
DataFrame.set_index : Set row labels.
DataFrame.reset_index : Remove row labels or move them to new columns.
DataFrame.reindex_like : Change to same indices as other DataFrame.

Examples
-----
``DataFrame.reindex`` supports two calling conventions

* ``(index=index_labels, columns=column_labels, ...)``
* ``(labels, axis={'index', 'columns'}, ...)``

We *highly* recommend using keyword arguments to clarify your
intent.

Create a dataframe with some fictional data.

>>> index = ['Firefox', 'Chrome', 'Safari', 'IE10', 'Konqueror']
>>> df = pd.DataFrame({'http_status': [200, 200, 404, 404, 301],
...                      'response_time': [0.04, 0.02, 0.07, 0.08, 1.0]},
...                      index=index)

```

```
>>> df
      http_status  response_time
Firefox          200        0.04
Chrome           200        0.02
Safari            404        0.07
IE10              404        0.08
Konqueror         301       1.00

Create a new index and reindex the dataframe. By default
values in the new index that do not have corresponding
records in the dataframe are assigned ``NaN``.

>>> new_index = ['Safari', 'Iceweasel', 'Comodo Dragon', 'IE10',
...                 'Chrome']
>>> df.reindex(new_index)
      http_status  response_time
Safari           404.0        0.07
Iceweasel         NaN         NaN
Comodo Dragon    NaN         NaN
IE10             404.0        0.08
Chrome            200.0        0.02
```

We can fill in the missing values by passing a value to the keyword ``fill_value``. Because the index is not monotonically increasing or decreasing, we cannot use arguments to the keyword ``method`` to fill the ``NaN`` values.

```
>>> df.reindex(new_index, fill_value=0)
      http_status  response_time
Safari           404        0.07
Iceweasel         0         0.00
Comodo Dragon    0         0.00
IE10             404        0.08
Chrome            200        0.02

>>> df.reindex(new_index, fill_value='missing')
      http_status  response_time
Safari           404        0.07
Iceweasel        missing     missing
Comodo Dragon   missing     missing
IE10             404        0.08
Chrome            200        0.02
```

We can also reindex the columns.

```
>>> df.reindex(columns=['http_status', 'user_agent'])
      http_status  user_agent
Firefox          200        NaN
Chrome           200        NaN
Safari            404        NaN
IE10              404        NaN
Konqueror         301        NaN
```

Or we can use "axis-style" keyword arguments

```
>>> df.reindex(['http_status', 'user_agent'], axis="columns")
      http_status  user_agent
Firefox          200        NaN
Chrome           200        NaN
Safari            404        NaN
IE10              404        NaN
Konqueror         301        NaN
```

To further illustrate the filling functionality in ``reindex``, we will create a dataframe with a monotonically increasing index (for example, a sequence of dates).

```
>>> date_index = pd.date_range('1/1/2010', periods=6, freq='D')
>>> df2 = pd.DataFrame({'prices': [100, 101, np.nan, 100, 89, 88]}, 
...                  index=date_index)
>>> df2
      prices
2010-01-01  100.0
2010-01-02  101.0
2010-01-03    NaN
2010-01-04  100.0
2010-01-05   89.0
2010-01-06   88.0
```

Suppose we decide to expand the dataframe to cover a wider date range.

```
>>> date_index2 = pd.date_range('12/29/2009', periods=10, freq='D')
>>> df2.reindex(date_index2)
      prices
2009-12-29    NaN
2009-12-30    NaN
2009-12-31    NaN
2010-01-01  100.0
2010-01-02  101.0
2010-01-03    NaN
2010-01-04  100.0
2010-01-05   89.0
2010-01-06   88.0
2010-01-07    NaN
```

The index entries that did not have a value in the original data frame (for example, '2009-12-29') are by default filled with ``NaN``. If desired, we can fill in the missing values using one of several options.

For example, to back-propagate the last valid value to fill the ``NaN`` values, pass ``bfill`` as an argument to the ``method`` keyword.

```
>>> df2.reindex(date_index2, method='bfill')
      prices
2009-12-29  100.0
2009-12-30  100.0
2009-12-31  100.0
2010-01-01  100.0
2010-01-02  101.0
2010-01-03    NaN
2010-01-04  100.0
2010-01-05   89.0
2010-01-06   88.0
2010-01-07    NaN
```

Please note that the ``NaN`` value present in the original dataframe (at index value 2010-01-03) will not be filled by any of the value propagation schemes. This is because filling while reindexing does not look at dataframe values, but only compares the original and desired indexes. If you do want to fill in the ``NaN`` values present in the original dataframe, use the ``fillna()`` method.

See the :ref:`user guide <basics.reindexing>` for more.

```

"""
# TODO: Decide if we care about having different examples for different
# kinds

# construct the args
axes, kwargs = self._construct_axes_from_arguments(args, kwargs)
method = missing.clean_reindex_fill_method(kwargs.pop("method", None))
level = kwargs.pop("level", None)
copy = kwargs.pop("copy", True)
limit = kwargs.pop("limit", None)
tolerance = kwargs.pop("tolerance", None)
fill_value = kwargs.pop("fill_value", None)

# Series.reindex doesn't use / need the axis kwarg
# We pop and ignore it here, to make writing Series/Frame generic code
# easier
kwargs.pop("axis", None)

if kwargs:
    raise TypeError(
        "reindex() got an unexpected keyword "
        f"argument '{list(kwargs.keys())[0]}'"
    )

self._consolidate_inplace()

# if all axes that are requested to reindex are equal, then only copy
# if indicated must have index names equal here as well as values
if all(
    self._get_axis(axis).identical(ax)
    for axis, ax in axes.items()
    if ax is not None
):
    if copy:
        return self.copy()
    return self

# check if we are a multi reindex
if self._needs_reindex_multi(axes, method, level):
    return self._reindex_multi(axes, copy, fill_value)

# perform the reindex on the axes
return self._reindex_axes(
    axes, level, limit, tolerance, method, fill_value, copy
).__finalize__(self, method="reindex")

def _reindex_axes(
    self: FrameOrSeries, axes, level, limit, tolerance, method, fill_value, copy
) -> FrameOrSeries:
    """Perform the reindex for all the axes."""
    obj = self
    for a in self._AXIS_ORDERS:
        labels = axes[a]
        if labels is None:
            continue

        ax = self._get_axis(a)
        new_index, indexer = ax.reindex(
            labels, level=level, limit=limit, tolerance=tolerance, method=method
        )

        axis = self._get_axis_number(a)
        obj = obj._reindex_with_indexers(

```

```

        {axis: [new_index, indexer]},
        fill_value=fill_value,
        copy=copy,
        allow_dups=False,
    )

return obj

def _needs_reindex_multi(self, axes, method, level) -> bool_t:
    """Check if we do need a multi reindex."""
    return (
        (com.count_not_none(*axes.values()) == self._AXIS_LEN)
        and method is None
        and level is None
        and not self._is_mixed_type
    )

def _reindex_multi(self, axes, copy, fill_value):
    raise AbstractMethodError(self)

def _reindex_with_indexers(
    self: FrameOrSeries,
    reindexers,
    fill_value=None,
    copy: bool_t = False,
    allow_dups: bool_t = False,
) -> FrameOrSeries:
    """allow_dups indicates an internal call here """
    # reindex doing multiple operations on different axes if indicated
    new_data = self._mgr
    for axis in sorted(reindexers.keys()):
        index, indexer = reindexers[axis]
        baxis = self._get_block_manager_axis(axis)

        if index is None:
            continue

        index = ensure_index(index)
        if indexer is not None:
            indexer = ensure_int64(indexer)

        # TODO: speed up on homogeneous DataFrame objects
        new_data = new_data.reindex_indexer(
            index,
            indexer,
            axis=baxis,
            fill_value=fill_value,
            allow_dups=allow_dups,
            copy=copy,
        )
    # If we've made a copy once, no need to make another one
    copy = False

    if copy and new_data is self._mgr:
        new_data = new_data.copy()

    return self._constructor(new_data).__finalize__(self)

def filter(
    self: FrameOrSeries,
    items=None,
    like: Optional[str] = None,
    regex: Optional[str] = None,

```

```
    axis=None,
) -> FrameOrSeries:
"""
Subset the dataframe rows or columns according to the specified index labels.

Note that this routine does not filter a dataframe on its
contents. The filter is applied to the labels of the index.

Parameters
-----
items : list-like
    Keep labels from axis which are in items.
like : str
    Keep labels from axis for which "like in label == True".
regex : str (regular expression)
    Keep labels from axis for which re.search(regex, label) == True.
axis : {0 or 'index', 1 or 'columns', None}, default None
    The axis to filter on, expressed either as an index (int)
    or axis name (str). By default this is the info axis,
    'index' for Series, 'columns' for DataFrame.

Returns
-----
same type as input object

See Also
-----
DataFrame.loc : Access a group of rows and columns
    by label(s) or a boolean array.

Notes
-----
The ``items``, ``like`` and ``regex`` parameters are
enforced to be mutually exclusive.

``axis`` defaults to the info axis that is used when indexing
with ``[]``.

Examples
-----
>>> df = pd.DataFrame(np.array(([1, 2, 3], [4, 5, 6])),
...                      index=['mouse', 'rabbit'],
...                      columns=['one', 'two', 'three'])
>>> df
   one  two  three
mouse    1    2    3
rabbit   4    5    6

>>> # select columns by name
>>> df.filter(items=['one', 'three'])
   one  three
mouse    1    3
rabbit   4    6

>>> # select columns by regular expression
>>> df.filter(regex='e$', axis=1)
   one  three
mouse    1    3
rabbit   4    6

>>> # select rows containing 'bbi'
>>> df.filter(like='bbi', axis=0)
   one  two  three
```

```

rabbit      4      5      6
"""
nkw = com.count_not_none(items, like, regex)
if nkw > 1:
    raise TypeError(
        "Keyword arguments `items`, `like`, or `regex` "
        "are mutually exclusive"
    )

if axis is None:
    axis = self._info_axis_name
labels = self._get_axis(axis)

if items is not None:
    name = self._get_axis_name(axis)
    return self.reindex(**{name: [r for r in items if r in labels]})

elif like:

    def f(x):
        return like in ensure_str(x)

    values = labels.map(f)
    return self.loc(axis=axis)[values]

elif regex:

    def f(x):
        return matcher.search(ensure_str(x)) is not None

    matcher = re.compile(regex)
    values = labels.map(f)
    return self.loc(axis=axis)[values]

else:
    raise TypeError("Must pass either `items`, `like`, or `regex`")

def head(self: FrameOrSeries, n: int = 5) -> FrameOrSeries:
"""
Return the first `n` rows.

This function returns the first `n` rows for the object based
on position. It is useful for quickly testing if your object
has the right type of data in it.

For negative values of `n`, this function returns all rows except
the last `n` rows, equivalent to ``df[:-n]``.

Parameters
-----
n : int, default 5
    Number of rows to select.

Returns
-----
same type as caller
    The first `n` rows of the caller object.

See Also
-----
DataFrame.tail: Returns the last `n` rows.

Examples
-----
>>> df = pd.DataFrame({'animal': ['alligator', 'bee', 'falcon', 'lion',
...                                'monkey', 'parrot', 'shark', 'whale', 'zebra']})

```

```

>>> df
      animal
0  alligator
1      bee
2    falcon
3      lion
4    monkey
5   parrot
6    shark
7    whale
8    zebra

Viewing the first 5 lines

>>> df.head()
      animal
0  alligator
1      bee
2    falcon
3      lion
4    monkey

Viewing the first `n` lines (three in this case)

>>> df.head(3)
      animal
0  alligator
1      bee
2    falcon

For negative values of `n`

>>> df.head(-3)
      animal
0  alligator
1      bee
2    falcon
3      lion
4    monkey
5   parrot
"""
return self.iloc[:n]

def tail(self: FrameOrSeries, n: int = 5) -> FrameOrSeries:
    """
    Return the last `n` rows.

    This function returns last `n` rows from the object based on
    position. It is useful for quickly verifying data, for example,
    after sorting or appending rows.

    For negative values of `n`, this function returns all rows except
    the first `n` rows, equivalent to ``df[n:]``.

    Parameters
    -----
    n : int, default 5
        Number of rows to select.

    Returns
    -----
    type of caller
        The last `n` rows of the caller object.

```

```
See Also
-----
DataFrame.head : The first `n` rows of the caller object.

Examples
-----
>>> df = pd.DataFrame({'animal': ['alligator', 'bee', 'falcon', 'lion',
...                                'monkey', 'parrot', 'shark', 'whale', 'zebra']})
>>> df
   animal
0 alligator
1      bee
2    falcon
3      lion
4    monkey
5    parrot
6      shark
7     whale
8     zebra

Viewing the last 5 lines

>>> df.tail()
   animal
4    monkey
5    parrot
6      shark
7     whale
8     zebra

Viewing the last `n` lines (three in this case)

>>> df.tail(3)
   animal
6    shark
7     whale
8     zebra

For negative values of `n`

>>> df.tail(-3)
   animal
3      lion
4    monkey
5    parrot
6      shark
7     whale
8     zebra
"""
if n == 0:
    return self.iloc[0:0]
return self.iloc[-n:]

def sample(
    self: FrameOrSeries,
    n=None,
    frac=None,
    replace=False,
    weights=None,
    random_state=None,
    axis=None,
) -> FrameOrSeries:
```

```
"""
Return a random sample of items from an axis of object.

You can use `random_state` for reproducibility.

Parameters
-----
n : int, optional
    Number of items from axis to return. Cannot be used with `frac`.
    Default = 1 if `frac` = None.
frac : float, optional
    Fraction of axis items to return. Cannot be used with `n`.
replace : bool, default False
    Allow or disallow sampling of the same row more than once.
weights : str or ndarray-like, optional
    Default 'None' results in equal probability weighting.
    If passed a Series, will align with target object on index. Index
    values in weights not found in sampled object will be ignored and
    index values in sampled object not in weights will be assigned
    weights of zero.
    If called on a DataFrame, will accept the name of a column
    when axis = 0.
    Unless weights are a Series, weights must be same length as axis
    being sampled.
    If weights do not sum to 1, they will be normalized to sum to 1.
    Missing values in the weights column will be treated as zero.
    Infinite values not allowed.
random_state : int, array-like, BitGenerator, np.random.RandomState, optional
    If int, array-like, or BitGenerator (NumPy>=1.17), seed for
    random number generator
    If np.random.RandomState, use as numpy RandomState object.

..versionchanged:: 1.1.0

    array-like and BitGenerator (for NumPy>=1.17) object now passed to
    np.random.RandomState() as seed

axis : {0 or 'index', 1 or 'columns', None}, default None
    Axis to sample. Accepts axis number or name. Default is stat axis
    for given data type (0 for Series and DataFrames).

Returns
-----
Series or DataFrame
    A new object of same type as caller containing `n` items randomly
    sampled from the caller object.

See Also
-----
numpy.random.choice: Generates a random sample from a given 1-D numpy
array.

Notes
-----
If `frac` > 1, `replacement` should be set to `True`.

Examples
-----
>>> df = pd.DataFrame({'num_legs': [2, 4, 8, 0],
...                     'num_wings': [2, 0, 0, 0],
...                     'num_specimen_seen': [10, 2, 1, 8]},
...                     index=['falcon', 'dog', 'spider', 'fish'])
>>> df
```

| | num_legs | num_wings | num_specimen_seen |
|--------|----------|-----------|-------------------|
| falcon | 2 | 2 | 10 |
| dog | 4 | 0 | 2 |
| spider | 8 | 0 | 1 |
| fish | 0 | 0 | 8 |

Extract 3 random elements from the ``Series`` ``df['num_legs']``:
Note that we use `random_state` to ensure the reproducibility of
the examples.

```
>>> df['num_legs'].sample(n=3, random_state=1)
fish      0
spider    8
falcon    2
Name: num_legs, dtype: int64
```

A random 50% sample of the ``DataFrame`` with replacement:

```
>>> df.sample(frac=0.5, replace=True, random_state=1)
      num_legs  num_wings  num_specimen_seen
dog          4          0                  2
fish         0          0                  8
```

An upsample sample of the ``DataFrame`` with replacement:
Note that `replace` parameter has to be `True` for `frac` parameter > 1.

```
>>> df.sample(frac=2, replace=True, random_state=1)
      num_legs  num_wings  num_specimen_seen
dog          4          0                  2
fish         0          0                  8
falcon       2          2                 10
falcon       2          2                 10
fish         0          0                  8
dog          4          0                  2
fish         0          0                  8
dog          4          0                  2
```

Using a DataFrame column as weights. Rows with larger value in the
`num_specimen_seen` column are more likely to be sampled.

```
>>> df.sample(n=2, weights='num_specimen_seen', random_state=1)
      num_legs  num_wings  num_specimen_seen
falcon       2          2                 10
fish         0          0                  8
"""
if axis is None:
    axis = self._stat_axis_number

axis = self._get_axis_number(axis)
axis_length = self.shape[axis]

# Process random_state argument
rs = com.random_state(random_state)

# Check weights for compliance
if weights is not None:

    # If a series, align with frame
    if isinstance(weights, ABCSeries):
        weights = weights.reindex(self.axes[axis])

    # Strings acceptable if a dataframe and axis = 0
    if isinstance(weights, str):
```

```

        if isinstance(self, ABCDataFrame):
            if axis == 0:
                try:
                    weights = self[weights]
                except KeyError as err:
                    raise KeyError(
                        "String passed to weights not a valid column"
                    ) from err
            else:
                raise ValueError(
                    "Strings can only be passed to "
                    "weights when sampling from rows on "
                    "a DataFrame"
                )
        else:
            raise ValueError(
                "Strings cannot be passed as weights "
                "when sampling from a Series."
            )

    weights = pd.Series(weights, dtype="float64")

    if len(weights) != axis_length:
        raise ValueError(
            "Weights and axis to be sampled must be of same length"
        )

    if (weights == np.inf).any() or (weights == -np.inf).any():
        raise ValueError("weight vector may not include `inf` values")

    if (weights < 0).any():
        raise ValueError("weight vector many not include negative values")

    # If has nan, set to zero.
    weights = weights.fillna(0)

    # Renormalize if don't sum to 1
    if weights.sum() != 1:
        if weights.sum() != 0:
            weights = weights / weights.sum()
        else:
            raise ValueError("Invalid weights: weights sum to zero")

    weights = weights.values

    # If no frac or n, default to n=1.
    if n is None and frac is None:
        n = 1
    elif frac is not None and frac > 1 and not replace:
        raise ValueError(
            "Replace has to be set to `True` when "
            "upsampling the population `frac` > 1."
        )
    elif n is not None and frac is None and n % 1 != 0:
        raise ValueError("Only integers accepted as `n` values")
    elif n is None and frac is not None:
        n = int(round(frac * axis_length))
    elif n is not None and frac is not None:
        raise ValueError("Please enter a value for `frac` OR `n`, not both")

    # Check for negative sizes
    if n < 0:
        raise ValueError(

```

```

        "A negative number of rows requested. Please provide positive value."
    )

locs = rs.choice(axis_length, size=n, replace=replace, p=weights)
return self.take(locs, axis=axis)

/shared_docs[
    "pipe"
] = r"""
    Apply func(self, \*args, \*\*kwargs).

Parameters
-----
func : function
    Function to apply to the %(klass)s.
    ``args``, and ``kwargs`` are passed into ``func``.
    Alternatively a ``(``callable, data_keyword``)`` tuple where
    ``data_keyword`` is a string indicating the keyword of
    ``callable`` that expects the %(klass)s.
args : iterable, optional
    Positional arguments passed into ``func``.
kwargs : mapping, optional
    A dictionary of keyword arguments passed into ``func``.

Returns
-----
object : the return type of ``func``.

See Also
-----
DataFrame.apply : Apply a function along input axis of DataFrame.
DataFrame.applymap : Apply a function elementwise on a whole DataFrame.
Series.map : Apply a mapping correspondence on a
    :class:`~pandas.Series`.

Notes
-----
Use ``.pipe`` when chaining together functions that expect
Series, DataFrames or GroupBy objects. Instead of writing

>>> func(g(h(df), arg1=a), arg2=b, arg3=c)  # doctest: +SKIP

You can write

>>> (df.pipe(h)
...     .pipe(g, arg1=a)
...     .pipe(func, arg2=b, arg3=c)
... )  # doctest: +SKIP

If you have a function that takes the data as (say) the second
argument, pass a tuple indicating which keyword expects the
data. For example, suppose ``f`` takes its data as ``arg2``:

>>> (df.pipe(h)
...     .pipe(g, arg1=a)
...     .pipe((func, 'arg2'), arg1=a, arg3=c)
... )  # doctest: +SKIP
"""

@Appender(_shared_docs["pipe"] % _shared_doc_kwargs)
def pipe(self, func, *args, **kwargs):
    return com.pipe(self, func, *args, **kwargs)

```

```

_shared_docs["aggregate"] = dedent(
    """
Aggregate using one or more operations over the specified axis.
%(versionadded)s
Parameters
-----
func : function, str, list or dict
    Function to use for aggregating the data. If a function, must either
    work when passed a %(klass)s or when passed to %(klass)s.apply.

    Accepted combinations are:

        - function
        - string function name
        - list of functions and/or function names, e.g. ``[np.sum, 'mean']``
        - dict of axis labels -> functions, function names or list of such.
%(axis)s
*args
    Positional arguments to pass to `func`.
**kwargs
    Keyword arguments to pass to `func`.

Returns
-----
scalar, Series or DataFrame

    The return can be:

        * scalar : when Series.agg is called with single function
        * Series : when DataFrame.agg is called with a single function
        * DataFrame : when DataFrame.agg is called with several functions

    Return scalar, Series or DataFrame.
%(see_also)s
Notes
-----
`agg` is an alias for `aggregate`. Use the alias.

A passed user-defined-function will be passed a Series for evaluation.
%(examples)s"""
)

_shared_docs[
    "transform"
] = """
Call ``func`` on self producing a %(klass)s with transformed values.

Produced %(klass)s will have same axis length as self.

Parameters
-----
func : function, str, list or dict
    Function to use for transforming the data. If a function, must either
    work when passed a %(klass)s or when passed to %(klass)s.apply.

    Accepted combinations are:

        - function
        - string function name
        - list of functions and/or function names, e.g. ``[np.exp, 'sqrt']``
        - dict of axis labels -> functions, function names or list of such.
%(axis)s
*args

```

```
    Positional arguments to pass to `func`.  
**kwargs  
    Keyword arguments to pass to `func`.  
  
Returns  
-----  
%(klass)s  
    A %(klass)s that must have the same length as self.  
  
Raises  
-----  
ValueError : If the returned %(klass)s has a different length than self.  
  
See Also  
-----  
%(klass)s.agg : Only perform aggregating type operations.  
%(klass)s.apply : Invoke function on a %(klass)s.  
  
Examples  
-----  
>>> df = pd.DataFrame({'A': range(3), 'B': range(1, 4)})  
>>> df  
      A   B  
0    0   1  
1    1   2  
2    2   3  
>>> df.transform(lambda x: x + 1)  
      A   B  
0    1   2  
1    2   3  
2    3   4
```

Even though the resulting %(klass)s must have the same length as the input %(klass)s, it is possible to provide several input functions:

```
>>> s = pd.Series(range(3))  
>>> s  
0    0  
1    1  
2    2  
dtype: int64  
>>> s.transform([np.sqrt, np.exp])  
      sqrt        exp  
0  0.000000  1.000000  
1  1.000000  2.718282  
2  1.414214  7.389056  
"""  
  
# -----  
# Attribute access  
  
def __finalize__(  
    self: FrameOrSeries, other, method: Optional[str] = None, **kwargs  
) -> FrameOrSeries:  
    """  
        Propagate metadata from other to self.  
  
    Parameters  
    -----  
    other : the object from which to get the attributes that we are going  
            to propagate  
    method : str, optional  
            A passed method name providing context on where ``__finalize__``
```

```

was called.

.. warning:

    The value passed as `method` are not currently considered
    stable across pandas releases.

"""
if isinstance(other, NDFrame):
    for name in other.attrs:
        self.attrs[name] = other.attrs[name]
    # For subclasses using _metadata.
    for name in self._metadata:
        assert isinstance(name, str)
        object.__setattr__(self, name, getattr(other, name, None))
return self

def __getattr__(self, name: str):
    """
    After regular attribute access, try looking up the name
    This allows simpler access to columns for interactive use.
    """
    # Note: obj.x will always call obj.__getattribute__('x') prior to
    # calling obj.__getattr__('x').
    if (
        name in self._internal_names_set
        or name in self._metadata
        or name in self._accessors
    ):
        return object.__getattribute__(self, name)
    else:
        if self._info_axis._can_hold_identifiers_and_holds_name(name):
            return self[name]
        return object.__getattribute__(self, name)

def __setattr__(self, name: str, value) -> None:
    """
    After regular attribute access, try setting the name
    This allows simpler access to columns for interactive use.
    """
    # first try regular attribute access via __getattribute__, so that
    # e.g. ``obj.x`` and ``obj.x = 4`` will always reference/modify
    # the same attribute.

    try:
        object.__getattribute__(self, name)
        return object.__setattr__(self, name, value)
    except AttributeError:
        pass

    # if this fails, go on to more involved attribute setting
    # (note that this matches __getattr__, above).
    if name in self._internal_names_set:
        object.__setattr__(self, name, value)
    elif name in self._metadata:
        object.__setattr__(self, name, value)
    else:
        try:
            existing = getattr(self, name)
            if isinstance(existing, Index):
                object.__setattr__(self, name, value)
            elif name in self._info_axis:
                self[name] = value
            else:

```

```

        object.__setattr__(self, name, value)
    except (AttributeError, TypeError):
        if isinstance(self, ABCDataFrame) and (is_list_like(value)):
            warnings.warn(
                "Pandas doesn't allow columns to be "
                "created via a new attribute name - see "
                "'https://pandas.pydata.org/pandas-docs/'"
                "'stable/indexing.html#attribute-access'",
                stacklevel=2,
            )
    object.__setattr__(self, name, value)

def __dir__(self):
    """
    add the string-like attributes from the info_axis.
    If info_axis is a MultiIndex, it's first level values are used.
    """
    additions = {
        c
        for c in self._info_axis.unique(level=0)[:100]
        if isinstance(c, str) and c.isidentifier()
    }
    return super().__dir__().union(additions)

# -----
# Consolidation of internals

def __protect_consolidate(self, f):
    """
    Consolidate _mgr -- if the blocks have changed, then clear the
    cache
    """
    blocks_before = len(self._mgr.blocks)
    result = f()
    if len(self._mgr.blocks) != blocks_before:
        self._clear_item_cache()
    return result

def __consolidate_inplace(self) -> None:
    """Consolidate data in place and return None"""

    def f():
        self._mgr = self._mgr.consolidate()

    self.__protect_consolidate(f)

def __consolidate(self, inplace: bool_t = False):
    """
    Compute NDFrame with "consolidated" internals (data of each dtype
    grouped together in a single ndarray).

    Parameters
    -----
    inplace : bool, default False
        If False return new object, otherwise modify existing object.

    Returns
    -----
    consolidated : same type as caller
    """
    inplace = validate_bool_kwarg(inplace, "inplace")
    if inplace:
        self._consolidate_inplace()

```

```

else:
    f = lambda: self._mgr.consolidate()
    cons_data = self._protect_consolidate(f)
    return self._constructor(cons_data).__finalize__(self)

@property
def _is_mixed_type(self) -> bool_t:
    f = lambda: self._mgr.is_mixed_type
    return self._protect_consolidate(f)

@property
def _is_numeric_mixed_type(self) -> bool_t:
    f = lambda: self._mgr.is_numeric_mixed_type
    return self._protect_consolidate(f)

def _check_inplace_setting(self, value) -> bool_t:
    """ check whether we allow in-place setting with this type of value """
    if self._is_mixed_type:
        if not self._is_numeric_mixed_type:

            # allow an actual np.nan thru
            if is_float(value) and np.isnan(value):
                return True

            raise TypeError(
                "Cannot do inplace boolean setting on "
                "mixed-types with a non np.nan value"
            )

    return True

def _get_numeric_data(self):
    return self._constructor(self._mgr.get_numeric_data()).__finalize__(self)

def _get_bool_data(self):
    return self._constructor(self._mgr.get_bool_data()).__finalize__(self)

# -----
# Internal Interface Methods

@property
def values(self) -> np.ndarray:
    """
    Return a Numpy representation of the DataFrame.

    .. warning::

        We recommend using :meth:`DataFrame.to_numpy` instead.

    Only the values in the DataFrame will be returned, the axes labels
    will be removed.

    Returns
    ------
    numpy.ndarray
        The values of the DataFrame.

    See Also
    -----
    DataFrame.to_numpy : Recommended alternative to this method.
    DataFrame.index : Retrieve the index labels.
    DataFrame.columns : Retrieving the column names.

```

```
Notes
```

```
----
```

```
The dtype will be a lower-common-denominator dtype (implicit upcasting); that is to say if the dtypes (even of numeric types) are mixed, the one that accommodates all will be chosen. Use this with care if you are not dealing with the blocks.
```

```
e.g. If the dtypes are float16 and float32, dtype will be upcast to float32. If dtypes are int32 and uint8, dtype will be upcast to int32. By :func:`numpy.find_common_type` convention, mixing int64 and uint64 will result in a float64 dtype.
```

```
Examples
```

```
-----
```

```
A DataFrame where all columns are the same type (e.g., int64) results in an array of the same type.
```

```
>>> df = pd.DataFrame({'age': [3, 29],  
...                      'height': [94, 170],  
...                      'weight': [31, 115]})  
>>> df  
    age  height  weight  
0     3      94      31  
1    29     170     115  
>>> df.dtypes  
age        int64  
height     int64  
weight     int64  
dtype: object  
>>> df.values  
array([[ 3,  94,  31],  
       [ 29, 170, 115]])
```

```
A DataFrame with mixed type columns(e.g., str/object, int64, float32) results in an ndarray of the broadest type that accommodates these mixed types (e.g., object).
```

```
>>> df2 = pd.DataFrame([('parrot', 24.0, 'second'),  
...                      ('lion', 80.5, 1),  
...                      ('monkey', np.nan, None)],  
...                      columns=('name', 'max_speed', 'rank'))  
>>> df2.dtypes  
name          object  
max_speed    float64  
rank          object  
dtype: object  
>>> df2.values  
array([['parrot', 24.0, 'second'],  
       ['lion', 80.5, 1],  
       ['monkey', nan, None]], dtype=object)  
"""\n    self._consolidate_inplace()  
    return self._mgr.asarray(transpose=self._AXIS_REVERSED)
```

```
@property
```

```
def _values(self) -> np.ndarray:  
    """internal implementation"""  
    return self.values
```

```
@property
```

```
def dtypes(self):  
    """  
    Return the dtypes in the DataFrame.
```

This returns a Series with the data type of each column.
The result's index is the original DataFrame's columns. Columns
with mixed types are stored with the ``object`` dtype. See
:ref:`the User Guide <basics.dtypes>` for more.

Returns

pandas.Series

The data type of each column.

Examples

```
>>> df = pd.DataFrame({'float': [1.0],
...                      'int': [1],
...                      'datetime': [pd.Timestamp('20180310')],
...                      'string': ['foo']})
>>> df.dtypes
float            float64
int             int64
datetime    datetime64[ns]
string          object
dtype: object
"""
from pandas import Series
```

```
return Series(self._mgr.get_dtypes(), index=self._info_axis, dtype=np.object_)
```

```
def _to_dict_of_blocks(self, copy: bool_t = True):
    """
    Return a dict of dtype -> Constructor Types that
    each is a homogeneous dtype.
```

Internal ONLY

"""
 return {
 k: self._constructor(v).finalize_(self)
 for k, v, in self._mgr.to_dict(copy=copy).items()
 }

```
def astype(
    self: FrameOrSeries, dtype, copy: bool_t = True, errors: str = "raise"
) -> FrameOrSeries:
    """
    Cast a pandas object to a specified dtype ``dtype``.
```

Parameters

dtype : data type, or dict of column name -> data type
Use a numpy.dtype or Python type to cast entire pandas object to
the same type. Alternatively, use {col: dtype, ...}, where col is a
column label and dtype is a numpy.dtype or Python type to cast one
or more of the DataFrame's columns to column-specific types.

copy : bool, default True

Return a copy when ``copy=True`` (be very careful setting
``copy=False`` as changes to values then may propagate to other
pandas objects).

errors : {'raise', 'ignore'}, default 'raise'

Control raising of exceptions on invalid data for provided dtype.

- ``raise`` : allow exceptions to be raised

- ``ignore`` : suppress exceptions. On error return original object.

```
Returns
-----
casted : same type as caller

See Also
-----
to_datetime : Convert argument to datetime.
to_timedelta : Convert argument to timedelta.
to_numeric : Convert argument to a numeric type.
numpy.ndarray.astype : Cast a numpy array to a specified type.
```

Examples

```
-----
```

```
Create a DataFrame:
```

```
>>> d = {'col1': [1, 2], 'col2': [3, 4]}
>>> df = pd.DataFrame(data=d)
>>> df.dtypes
col1    int64
col2    int64
dtype: object
```

```
Cast all columns to int32:
```

```
>>> df.astype('int32').dtypes
col1    int32
col2    int32
dtype: object
```

```
Cast col1 to int32 using a dictionary:
```

```
>>> df.astype({'col1': 'int32'}).dtypes
col1    int32
col2    int64
dtype: object
```

```
Create a series:
```

```
>>> ser = pd.Series([1, 2], dtype='int32')
>>> ser
0    1
1    2
dtype: int32
>>> ser.astype('int64')
0    1
1    2
dtype: int64
```

```
Convert to categorical type:
```

```
>>> ser.astype('category')
0    1
1    2
dtype: category
Categories (2, int64): [1, 2]
```

```
Convert to ordered categorical type with custom ordering:
```

```
>>> cat_dtype = pd.api.types.CategoricalDtype(
...     categories=[2, 1], ordered=True)
>>> ser.astype(cat_dtype)
0    1
1    2
```

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

Note that using ``copy=False`` and changing data on a new pandas object may propagate changes:

```
>>> s1 = pd.Series([1, 2])
>>> s2 = s1.astype('int64', copy=False)
>>> s2[0] = 10
>>> s1 # note that s1[0] has changed too
0    10
1     2
dtype: int64
```

Create a series of dates:

```
>>> ser_date = pd.Series(pd.date_range('20200101', periods=3))
>>> ser_date
0    2020-01-01
1    2020-01-02
2    2020-01-03
dtype: datetime64[ns]
```

Datetimes are localized to UTC first before converting to the specified timezone:

```
>>> ser_date.astype('datetime64[ns, US/Eastern]')
0    2019-12-31 19:00:00-05:00
1    2020-01-01 19:00:00-05:00
2    2020-01-02 19:00:00-05:00
dtype: datetime64[ns, US/Eastern]
"""

if is_dict_like(dtype):
    if self.ndim == 1: # i.e. Series
        if len(dtype) > 1 or self.name not in dtype:
            raise KeyError(
                "Only the Series name can be used for "
                "the key in Series dtype mappings."
            )
        new_type = dtype[self.name]
        return self.astype(new_type, copy, errors)

    for col_name in dtype.keys():
        if col_name not in self:
            raise KeyError(
                "Only a column name can be used for the "
                "key in a dtype mappings argument."
            )
    results = []
    for col_name, col in self.items():
        if col_name in dtype:
            results.append(
                col.astype(dtype=dtype[col_name], copy=copy, errors=errors)
            )
        else:
            results.append(col.copy() if copy else col)

elif is_extension_array_dtype(dtype) and self.ndim > 1:
    # GH 18099/22869: columnwise conversion to extension dtype
    # GH 24704: use iloc to handle duplicate column names
    results = [
        self.iloc[:, i].astype(dtype, copy=copy)
        for i in range(len(self.columns))
```

```
]

else:
    # else, only a single dtype is given
    new_data = self._mgr.astype(dtype=dtype, copy=copy, errors=errors,)
    return self._constructor(new_data).__finalize__(self, method="astype")

# GH 19920: retain column metadata after concat
result = pd.concat(results, axis=1, copy=False)
result.columns = self.columns
return result

def copy(self: FrameOrSeries, deep: bool_t = True) -> FrameOrSeries:
    """
    Make a copy of this object's indices and data.

    When ``deep=True`` (default), a new object will be created with a
    copy of the calling object's data and indices. Modifications to
    the data or indices of the copy will not be reflected in the
    original object (see notes below).

    When ``deep=False``, a new object will be created without copying
    the calling object's data or index (only references to the data
    and index are copied). Any changes to the data of the original
    will be reflected in the shallow copy (and vice versa).
    """

    Parameters
    -----
    deep : bool, default True
        Make a deep copy, including a copy of the data and the indices.
        With ``deep=False`` neither the indices nor the data are copied.

    Returns
    -----
    copy : Series or DataFrame
        Object type matches caller.

    Notes
    -----
    When ``deep=True``, data is copied but actual Python objects
    will not be copied recursively, only the reference to the object.
    This is in contrast to `copy.deepcopy` in the Standard Library,
    which recursively copies object data (see examples below).

    While ``Index`` objects are copied when ``deep=True``, the underlying
    numpy array is not copied for performance reasons. Since ``Index`` is
    immutable, the underlying data can be safely shared and a copy
    is not needed.

    Examples
    -----
    >>> s = pd.Series([1, 2], index=["a", "b"])
    >>> s
    a    1
    b    2
    dtype: int64

    >>> s_copy = s.copy()
    >>> s_copy
    a    1
    b    2
    dtype: int64
```

```

**Shallow copy versus default (deep) copy:**

>>> s = pd.Series([1, 2], index=["a", "b"])
>>> deep = s.copy()
>>> shallow = s.copy(deep=False)

Shallow copy shares data and index with original.

>>> s is shallow
False
>>> s.values is shallow.values and s.index is shallow.index
True

Deep copy has own copy of data and index.

>>> s is deep
False
>>> s.values is deep.values or s.index is deep.index
False

```

Updates to the data shared by shallow copy and original is reflected in both; deep copy remains unchanged.

```

>>> s[0] = 3
>>> shallow[1] = 4
>>> s
a    3
b    4
dtype: int64
>>> shallow
a    3
b    4
dtype: int64
>>> deep
a    1
b    2
dtype: int64

```

Note that when copying an object containing Python objects, a deep copy will copy the data, but will not do so recursively. Updating a nested data object will be reflected in the deep copy.

```

>>> s = pd.Series([[1, 2], [3, 4]])
>>> deep = s.copy()
>>> s[0][0] = 10
>>> s
0    [10, 2]
1    [3, 4]
dtype: object
>>> deep
0    [10, 2]
1    [3, 4]
dtype: object
"""
data = self._mgr.copy(deep=deep)
self._clear_item_cache()
return self._constructor(data).__finalize__(self, method="copy")

def __copy__(self: FrameOrSeries, deep: bool_t = True) -> FrameOrSeries:
    return self.copy(deep=deep)

def __deepcopy__(self: FrameOrSeries, memo=None) -> FrameOrSeries:
    """

```

```
Parameters
-----
memo, default None
    Standard signature. Unused
"""
return self.copy(deep=True)

def _convert(
    self: FrameOrSeries,
    datetime: bool_t = False,
    numeric: bool_t = False,
    timedelta: bool_t = False,
    coerce: bool_t = False,
) -> FrameOrSeries:
    """
    Attempt to infer better dtype for object columns

    Parameters
    -----
    datetime : bool, default False
        If True, convert to date where possible.
    numeric : bool, default False
        If True, attempt to convert to numbers (including strings), with
        unconvertible values becoming NaN.
    timedelta : bool, default False
        If True, convert to timedelta where possible.
    coerce : bool, default False
        If True, force conversion with unconvertible values converted to
        nulls (NaN or NaT).

    Returns
    -----
    converted : same as input object
    """
    validate_bool_kwarg(datetime, "datetime")
    validate_bool_kwarg(numeric, "numeric")
    validate_bool_kwarg(timedelta, "timedelta")
    validate_bool_kwarg(coerce, "coerce")
    return self._constructor(
        self._mgr.convert(
            datetime=datetime,
            numeric=numeric,
            timedelta=timedelta,
            coerce=coerce,
            copy=True,
        )
    ).__finalize__(self)

def infer_objects(self: FrameOrSeries) -> FrameOrSeries:
    """
    Attempt to infer better dtypes for object columns.

    Attempts soft conversion of object-dtyped
    columns, leaving non-object and unconvertible
    columns unchanged. The inference rules are the
    same as during normal Series/DataFrame construction.

    Returns
    -----
    converted : same type as input object

    See Also
    -----
```

```
to_datetime : Convert argument to datetime.
to_timedelta : Convert argument to timedelta.
to_numeric : Convert argument to numeric type.
convert_dtypes : Convert argument to best possible dtype.

Examples
-----
>>> df = pd.DataFrame({"A": ["a", 1, 2, 3]})
>>> df = df.iloc[1:]
>>> df
   A
1  1
2  2
3  3

>>> df.dtypes
A    object
dtype: object

>>> df.infer_objects().dtypes
A    int64
dtype: object
"""
# numeric=False necessary to only soft convert;
# python objects will still be converted to
# native numpy numeric types
return self._constructor(
    self._mgr.convert(
        datetime=True, numeric=False, timedelta=True, coerce=False, copy=True
    )
).__finalize__(self, method="infer_objects")

def convert_dtypes(
    self: FrameOrSeries,
    infer_objects: bool_t = True,
    convert_string: bool_t = True,
    convert_integer: bool_t = True,
    convert_boolean: bool_t = True,
) -> FrameOrSeries:
"""
Convert columns to best possible dtypes using dtypes supporting ``pd.NA``.

.. versionadded:: 1.0.0

Parameters
-----
infer_objects : bool, default True
    Whether object dtypes should be converted to the best possible types.
convert_string : bool, default True
    Whether object dtypes should be converted to ``StringDtype()``.
convert_integer : bool, default True
    Whether, if possible, conversion can be done to integer extension types.
convert_boolean : bool, default True
    Whether object dtypes should be converted to ``BooleanDtype()``.

Returns
-----
Series or DataFrame
    Copy of input object with new dtype.

See Also
-----
infer_objects : Infer dtypes of objects.
```

```
to_datetime : Convert argument to datetime.  
to_timedelta : Convert argument to timedelta.  
to_numeric : Convert argument to a numeric type.
```

Notes

By default, ``convert_dtypes`` will attempt to convert a Series (or each Series in a DataFrame) to dtypes that support ``pd.NA``. By using the options ``convert_string``, ``convert_integer``, and ``convert_boolean``, it is possible to turn off individual conversions to ``StringDtype``, the integer extension types or ``BooleanDtype``, respectively.

For object-dtyped columns, if ``infer_objects`` is ``True``, use the inference rules as during normal Series/DataFrame construction. Then, if possible, convert to ``StringDtype``, ``BooleanDtype`` or an appropriate integer extension type, otherwise leave as ``object``.

If the dtype is integer, convert to an appropriate integer extension type.

If the dtype is numeric, and consists of all integers, convert to an appropriate integer extension type.

In the future, as new dtypes are added that support ``pd.NA``, the results of this method will change to support those new dtypes.

Examples

```
>>> df = pd.DataFrame(  
...     {  
...         "a": pd.Series([1, 2, 3], dtype=np.dtype("int32")),  
...         "b": pd.Series(["x", "y", "z"], dtype=np.dtype("O")),  
...         "c": pd.Series([True, False, np.nan], dtype=np.dtype("O")),  
...         "d": pd.Series(["h", "i", np.nan], dtype=np.dtype("O")),  
...         "e": pd.Series([10, np.nan, 20], dtype=np.dtype("float")),  
...         "f": pd.Series([np.nan, 100.5, 200], dtype=np.dtype("float")),  
...     }  
... )
```

Start with a DataFrame with default dtypes.

```
>>> df  
   a   b      c      d      e      f  
0  1   x    True     h  10.0    NaN  
1  2   y   False     i    NaN  100.5  
2  3   z     NaN    NaN  20.0  200.0
```

```
>>> df.dtypes  
a      int32  
b      object  
c      object  
d      object  
e    float64  
f    float64  
dtype: object
```

Convert the DataFrame to use best possible dtypes.

```
>>> dfn = df.convert_dtypes()  
>>> dfn  
   a   b      c      d      e      f  
0  1   x    True     h     10    NaN  
1  2   y   False     i    <NA>  100.5  
2  3   z    <NA>  <NA>     20  200.0
```

```

>>> dfn.dtypes
a      Int32
b      string
c    boolean
d      string
e     Int64
f   float64
dtype: object

Start with a Series of strings and missing data represented by ``np.nan``.

>>> s = pd.Series(["a", "b", np.nan])
>>> s
0    a
1    b
2    NaN
dtype: object

Obtain a Series with dtype ``StringDtype``.

>>> s.convert_dtypes()
0    a
1    b
2    <NA>
dtype: string
"""

if self.ndim == 1:
    return self._convert_dtypes(
        infer_objects, convert_string, convert_integer, convert_boolean
    )
else:
    results = [
        col._convert_dtypes(
            infer_objects, convert_string, convert_integer, convert_boolean
        )
        for col_name, col in self.items()
    ]
    result = pd.concat(results, axis=1, copy=False)
    return result

# -----
# Filling NA's

@doc(**_shared_doc_kwargs)
def fillna(
    self: FrameOrSeries,
    value=None,
    method=None,
    axis=None,
    inplace: bool_t = False,
    limit=None,
    downcast=None,
) -> Optional[FrameOrSeries]:
    """
    Fill NA/NaN values using the specified method.

    Parameters
    -----
    value : scalar, dict, Series, or DataFrame
        Value to use to fill holes (e.g. 0), alternately a
        dict/Series/DataFrame of values specifying which value to use for
        each index (for a Series) or column (for a DataFrame). Values not

```

```
    in the dict/Series/DataFrame will not be filled. This value cannot
    be a list.

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.

axis : {axes_single_arg}
    Axis along which to fill missing values.

inplace : bool, default False
    If True, fill in-place. Note: this will modify any
    other views on this object (e.g., a no-copy slice for a column in a
    DataFrame).

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. Must be greater than 0 if not None.

downcast : dict, default is None
    A dict of item->dtype of what to downcast if possible,
    or the string 'infer' which will try to downcast to an appropriate
    equal type (e.g. float64 to int64 if possible).

Returns
-----
{klass} or None
    Object with missing values filled or None if ``inplace=True``.
```

See Also

```
-----
interpolate : Fill NaN values using interpolation.
reindex : Conform object to new index.
asfreq : Convert TimeSeries to specified frequency.
```

Examples

```
-----
>>> df = pd.DataFrame([[np.nan, 2, np.nan, 0],
...                      [3, 4, np.nan, 1],
...                      [np.nan, np.nan, np.nan, 5],
...                      [np.nan, 3, np.nan, 4]],
...                      columns=list('ABCD'))
>>> df
   A    B    C    D
0  NaN  2.0  NaN  0
1  3.0  4.0  NaN  1
2  NaN  NaN  NaN  5
3  NaN  3.0  NaN  4
```

Replace all NaN elements with 0s.

```
>>> df.fillna(0)
   A    B    C    D
0  0.0  2.0  0.0  0
1  3.0  4.0  0.0  1
2  0.0  0.0  0.0  5
3  0.0  3.0  0.0  4
```

We can also propagate non-null values forward or backward.

```
>>> df.fillna(method='ffill')
   A    B    C    D
0  NaN  2.0  NaN  0
```

```
1 3.0 4.0 NaN 1
2 3.0 4.0 NaN 5
3 3.0 3.0 NaN 4
```

Replace all NaN elements in column 'A', 'B', 'C', and 'D', with 0, 1, 2, and 3 respectively.

```
>>> values = {'A': 0, 'B': 1, 'C': 2, 'D': 3}
>>> df.fillna(value=values)
   A    B    C    D
0  0.0  2.0  2.0  0
1  3.0  4.0  2.0  1
2  0.0  1.0  2.0  5
3  0.0  3.0  2.0  4
```

Only replace the first NaN element.

```
>>> df.fillna(value=values, limit=1)
   A    B    C    D
0  0.0  2.0  2.0  0
1  3.0  4.0  NaN  1
2  NaN  1.0  NaN  5
3  NaN  3.0  NaN  4
"""
inplace = validate_bool_kwarg(inplace, "inplace")
value, method = validate_fillna_kwargs(value, method)

self._consolidate_inplace()

# set the default here, so functions examining the signature
# can detect if something was set (e.g. in groupby) (GH9221)
if axis is None:
    axis = 0
axis = self._get_axis_number(axis)

if value is None:

    if self._is_mixed_type and axis == 1:
        if inplace:
            raise NotImplementedError()
        result = self.T.fillna(method=method, limit=limit).T

        # need to downcast here because of all of the transposes
        result._mgr = result._mgr.downcast()

        return result

    new_data = self._mgr.interpolate(
        method=method,
        axis=axis,
        limit=limit,
        inplace=inplace,
        coerce=True,
        downcast=downcast,
    )
else:
    if self.ndim == 1:
        if isinstance(value, (dict, ABCSeries)):
            value = create_series_with_explicit_dtype(
                value, dtype_if_empty=object
            )
        value = value.reindex(self.index, copy=False)
        value = value._values
```

```

        elif not is_list_like(value):
            pass
        else:
            raise TypeError(
                '"value" parameter must be a scalar, dict '
                'or Series, but you passed a '
                f'"{type(value).__name__}"'
            )

    new_data = self._mgr.fillna(
        value=value, limit=limit, inplace=inplace, downcast=downcast
    )

    elif isinstance(value, (dict, ABCSeries)):
        if axis == 1:
            raise NotImplementedError(
                "Currently only can fill "
                "with dict/Series column "
                "by column"
            )

    result = self if inplace else self.copy()
    for k, v in value.items():
        if k not in result:
            continue
        obj = result[k]
        obj.fillna(v, limit=limit, inplace=True, downcast=downcast)
    return result if not inplace else None

    elif not is_list_like(value):
        new_data = self._mgr.fillna(
            value=value, limit=limit, inplace=inplace, downcast=downcast
        )
    elif isinstance(value, ABCDataFrame) and self.ndim == 2:
        new_data = self.where(self.notna(), value)._data
    else:
        raise ValueError(f"invalid fill value with a {type(value)}")

    result = self._constructor(new_data)
    if inplace:
        return self._update_inplace(result)
    else:
        return result.__finalize__(self, method="fillna")

def ffill(
    self: FrameOrSeries,
    axis=None,
    inplace: bool_t = False,
    limit=None,
    downcast=None,
) -> Optional[FrameOrSeries]:
    """
    Synonym for :meth:`DataFrame.fillna` with ``method='ffill'``.

    Returns
    -----
    %(klass)s or None
        Object with missing values filled or None if ``inplace=True``.
    """
    return self.fillna(
        method="ffill", axis=axis, inplace=inplace, limit=limit, downcast=downcast
    )

```

```

def bfill(
    self: FrameOrSeries,
    axis=None,
    inplace: bool_t = False,
    limit=None,
    downcast=None,
) -> Optional[FrameOrSeries]:
    """
    Synonym for :meth:`DataFrame.fillna` with ``method='bfill'``.

    Returns
    -----
    %(klass)s or None
        Object with missing values filled or None if ``inplace=True``.
    """
    return self.fillna(
        method="bfill", axis=axis, inplace=inplace, limit=limit, downcast=downcast
    )

@doc(klass=_shared_doc_kwargs["klass"])
def replace(
    self,
    to_replace=None,
    value=None,
    inplace=False,
    limit=None,
    regex=False,
    method="pad",
):
    """
    Replace values given in `to_replace` with `value`.

    Values of the {klass} are replaced with other values dynamically.
    This differs from updating with ``.loc`` or ``.iloc``, which require
    you to specify a location to update with some value.

    Parameters
    -----
    to_replace : str, regex, list, dict, Series, int, float, or None
        How to find the values that will be replaced.

        * numeric, str or regex:
            - numeric: numeric values equal to `to_replace` will be
              replaced with `value`
            - str: string exactly matching `to_replace` will be replaced
              with `value`
            - regex: regexes matching `to_replace` will be replaced with
              `value`

        * list of str, regex, or numeric:
            - First, if `to_replace` and `value` are both lists, they
              **must** be the same length.
            - Second, if ``regex=True`` then all of the strings in **both**
              lists will be interpreted as regexes otherwise they will match
              directly. This doesn't matter much for `value` since there
              are only a few possible substitution regexes you can use.
            - str, regex and numeric rules apply as above.

        * dict:
            - Dicts can be used to specify different replacement values
    """

```

for different existing values. For example,
```{{'a': 'b', 'y': 'z'}}``` replaces the value 'a' with 'b' and  
'y' with 'z'. To use a dict in this way the `value`  
parameter should be `'None'`.

- For a DataFrame a dict can specify that different values  
should be replaced in different columns. For example,  
```{{'a': 1, 'b': 'z'}}``` looks for the value 1 in column 'a'  
and the value 'z' in column 'b' and replaces these values
with whatever is specified in `value`. The `value` parameter
should not be `'None'` in this case. You can treat this as a
special case of passing two lists except that you are
specifying the column to search in.
- For a DataFrame nested dictionaries, e.g.,
```{{'a': {{'b': np.nan}}}}```, are read as follows: look in column  
'a' for the value 'b' and replace it with NaN. The `value`  
parameter should be `'None'` to use a nested dict in this  
way. You can nest regular expressions as well. Note that  
column names (the top-level dictionary keys in a nested  
dictionary) **cannot** be regular expressions.

\* `None`:

- This means that the `regex` argument must be a string,  
compiled regular expression, or list, dict, ndarray or  
Series of such elements. If `value` is also `'None'` then  
this **must** be a nested dictionary or Series.

See the examples section for examples of each of these.

`value` : scalar, dict, list, str, regex, default `None`  
Value to replace any values matching `'to_replace'` with.  
For a DataFrame a dict of values can be used to specify which  
value to use for each column (columns not in the dict will not be  
filled). Regular expressions, strings and lists or dicts of such  
objects are also allowed.

`inplace` : bool, default `False`

If `True`, in place. Note: this will modify any  
other views on this object (e.g. a column from a DataFrame).  
Returns the caller if this is `True`.

`limit` : int, default `None`

Maximum size gap to forward or backward fill.

`regex` : bool or same types as `'to_replace'`, default `False`

Whether to interpret `'to_replace'` and/or `'value'` as regular  
expressions. If this is `'True'` then `'to_replace'` **must** be a  
string. Alternatively, this could be a regular expression or a  
list, dict, or array of regular expressions in which case  
`'to_replace'` must be `'None'`.

`method` : `{'pad', 'ffill', 'bfill', 'None'}`

The method to use when for replacement, when `'to_replace'` is a  
scalar, list or tuple and `'value'` is `'None'`.

.. versionchanged:: 0.23.0

Added to DataFrame.

Returns

-----

`{klass}`

Object after replacement.

Raises

-----

`AssertionError`

\* If `'regex'` is not a `'bool'` and `'to_replace'` is not  
`'None'`.

```
TypeError
 * If `to_replace` is not a scalar, array-like, ``dict``, or ``None``
 * If `to_replace` is a ``dict`` and `value` is not a ``list``,
 ``dict``, ``ndarray``, or ``Series``
 * If `to_replace` is ``None`` and `regex` is not compilable
 into a regular expression or is a list, dict, ndarray, or
 Series.
 * When replacing multiple ``bool`` or ``datetime64`` objects and
 the arguments to `to_replace` does not match the type of the
 value being replaced
```

#### ValueError

```
* If a ``list`` or an ``ndarray`` is passed to `to_replace` and
 `value` but they are not the same length.
```

#### See Also

```

{klass}.fillna : Fill NA values.
{klass}.where : Replace values based on boolean condition.
Series.str.replace : Simple string replacement.
```

#### Notes

```

* Regex substitution is performed under the hood with ``re.sub``. The
 rules for substitution for ``re.sub`` are the same.
* Regular expressions will only substitute on strings, meaning you
 cannot provide, for example, a regular expression matching floating
 point numbers and expect the columns in your frame that have a
 numeric dtype to be matched. However, if those floating point
 numbers *are* strings, then you can do this.
* This method has *a lot* of options. You are encouraged to experiment
 and play with this method to gain intuition about how it works.
* When dict is used as the `to_replace` value, it is like
 key(s) in the dict are the to_replace part and
 value(s) in the dict are the value parameter.
```

#### Examples

```

Scalar `to_replace` and `value`
```

```
>>> s = pd.Series([0, 1, 2, 3, 4])
>>> s.replace(0, 5)
0 5
1 1
2 2
3 3
4 4
dtype: int64

>>> df = pd.DataFrame({{'A': [0, 1, 2, 3, 4],
... 'B': [5, 6, 7, 8, 9],
... 'C': ['a', 'b', 'c', 'd', 'e']}})
>>> df.replace(0, 5)
 A B C
0 5 5 a
1 1 6 b
2 2 7 c
3 3 8 d
4 4 9 e

List-like `to_replace`
```

```

>>> df.replace([0, 1, 2, 3], 4)
 A B C
0 4 5 a
1 4 6 b
2 4 7 c
3 4 8 d
4 4 9 e

>>> df.replace([0, 1, 2, 3], [4, 3, 2, 1])
 A B C
0 4 5 a
1 3 6 b
2 2 7 c
3 1 8 d
4 4 9 e

>>> s.replace([1, 2], method='bfill')
0 0
1 3
2 3
3 3
4 4
dtype: int64

dict-like `to_replace`

>>> df.replace({{0: 10, 1: 100}})
 A B C
0 10 5 a
1 100 6 b
2 2 7 c
3 3 8 d
4 4 9 e

>>> df.replace({{'A': 0, 'B': 5}}, 100)
 A B C
0 100 100 a
1 1 6 b
2 2 7 c
3 3 8 d
4 4 9 e

>>> df.replace({{'A': {{0: 100, 4: 400}}}})
 A B C
0 100 5 a
1 1 6 b
2 2 7 c
3 3 8 d
4 400 9 e

Regular expression `to_replace`

>>> df = pd.DataFrame({{'A': ['bat', 'foo', 'bait'],
... 'B': ['abc', 'bar', 'xyz']}})
>>> df.replace(to_replace=r'^ba.$', value='new', regex=True)
 A B
0 new abc
1 foo new
2 bait xyz

>>> df.replace({{'A': r'^ba.$'}}, {'A': 'new'}, regex=True)
 A B

```

```

0 new abc
1 foo bar
2 bait xyz

>>> df.replace(regex=r'^ba.$', value='new')
 A B
0 new abc
1 foo new
2 bait xyz

>>> df.replace(regex={r'^ba.$': 'new', 'foo': 'xyz'})
 A B
0 new abc
1 xyz new
2 bait xyz

>>> df.replace(regex=[r'^ba.$', 'foo'], value='new')
 A B
0 new abc
1 new new
2 bait xyz

```

Note that when replacing multiple ``bool`` or ``datetime64`` objects, the data types in the `to\_replace` parameter must match the data type of the value being replaced:

```

>>> df = pd.DataFrame({{'A': [True, False, True],
... 'B': [False, True, False]}})
>>> df.replace({{'a string': 'new value', True: False}}) # raises
Traceback (most recent call last):
...
TypeError: Cannot compare types 'ndarray(dtype=bool)' and 'str'

```

This raises a ``TypeError`` because one of the ``dict`` keys is not of the correct type for replacement.

Compare the behavior of ``s.replace({{'a': None}})`` and ``s.replace('a', None)`` to understand the peculiarities of the `to\_replace` parameter:

```
>>> s = pd.Series([10, 'a', 'a', 'b', 'a'])
```

When one uses a dict as the `to\_replace` value, it is like the value(s) in the dict are equal to the `value` parameter.

``s.replace({{'a': None}})`` is equivalent to  
``s.replace(to\_replace={'a': None}, value=None, method=None)``:

```

>>> s.replace({{'a': None}})
0 10
1 None
2 None
3 b
4 None
dtype: object

```

When ``value=None`` and `to\_replace` is a scalar, list or tuple, `replace` uses the `method` parameter (default 'pad') to do the replacement. So this is why the 'a' values are being replaced by 10 in rows 1 and 2 and 'b' in row 4 in this case.

The command ``s.replace('a', None)`` is actually equivalent to  
``s.replace(to\_replace='a', value=None, method='pad')``:

```
>>> s.replace('a', None)
```

```

0 10
1 10
2 10
3 b
4 b
dtype: object
"""
if not (
 is_scalar(to_replace)
 or is_re_compilable(to_replace)
 or is_list_like(to_replace)
):
 raise TypeError(
 "Expecting 'to_replace' to be either a scalar, array-like, "
 "dict or None, got invalid type "
 f"{repr(type(to_replace)).__name__}"
)

inplace = validate_bool_kwarg(inplace, "inplace")
if not is_bool(regex) and to_replace is not None:
 raise AssertionError("'to_replace' must be 'None' if 'regex' is not a bool")

self._consolidate_inplace()

if value is None:
 # passing a single value that is scalar like
 # when value is None (GH5319), for compat
 if not is_dict_like(to_replace) and not is_dict_like(regex):
 to_replace = [to_replace]

 if isinstance(to_replace, (tuple, list)):
 if isinstance(self, ABCDataFrame):
 return self.apply(
 _single_replace, args=(to_replace, method, inplace, limit)
)
 return _single_replace(self, to_replace, method, inplace, limit)

 if not is_dict_like(to_replace):
 if not is_dict_like(regex):
 raise TypeError(
 'If "to_replace" and "value" are both None '
 'and "to_replace" is not a list, then '
 "regex must be a mapping"
)
 to_replace = regex
 regex = True

 items = list(to_replace.items())
 keys, values = zip(*items) if items else ([], [])
 are_mappings = [is_dict_like(v) for v in values]

 if any(are_mappings):
 if not all(are_mappings):
 raise TypeError(
 "If a nested mapping is passed, all values "
 "of the top level mapping must be mappings"
)
 # passed a nested dict/Series
 to_rep_dict = {}
 value_dict = {}

 for k, v in items:

```

```

 keys, values = list(zip(*v.items())) or ([], [])
 to_rep_dict[k] = list(keys)
 value_dict[k] = list(values)

 to_replace, value = to_rep_dict, value_dict
 else:
 to_replace, value = keys, values

 return self.replace(
 to_replace, value, inplace=inplace, limit=limit, regex=regex
)
else:

 # need a non-zero len on all axes
 if not self.size:
 return self

 if is_dict_like(to_replace):
 if is_dict_like(value): # {'A' : NA} -> {'A' : 0}
 # Note: Checking below for `in foo.keys()` instead of
 # `in foo` is needed for when we have a Series and not dict
 mapping = {
 col: (to_replace[col], value[col])
 for col in to_replace.keys()
 if col in value.keys() and col in self
 }
 return self._replace_columnwise(mapping, inplace, regex)

 # {'A': NA} -> 0
 elif not is_list_like(value):
 # Operate column-wise
 if self.ndim == 1:
 raise ValueError(
 "Series.replace cannot use dict-like to_replace "
 "and non-None value"
)
 mapping = {
 col: (to_replace, value) for col, to_replace in to_replace.items()
 }
 return self._replace_columnwise(mapping, inplace, regex)
 else:
 raise TypeError("value argument must be scalar, dict, or Series")

elif is_list_like(to_replace): # [NA, ''] -> [0, 'missing']
 if is_list_like(value):
 if len(to_replace) != len(value):
 raise ValueError(
 f"Replacement lists must match in length. "
 f"Expecting {len(to_replace)} got {len(value)} "
)

 new_data = self._mgr.replace_list(
 src_list=to_replace,
 dest_list=value,
 inplace=inplace,
 regex=regex,
)

 else: # [NA, ''] -> 0
 new_data = self._mgr.replace(
 to_replace=to_replace, value=value, inplace=inplace, regex=regex
)

```

```

 elif to_replace is None:
 if not (
 is_re_compilable(regex)
 or is_list_like(regex)
 or is_dict_like(regex)
):
 raise TypeError(
 f"'regex' must be a string or a compiled regular expression "
 f"or a list or dict of strings or regular expressions, "
 f"you passed a {repr(type(regex).__name__)}"
)
 return self.replace(
 regex, value, inplace=inplace, limit=limit, regex=True
)
else:

 # dest iterable dict-like
 if is_dict_like(value): # NA -> {'A' : 0, 'B' : -1}
 # Operate column-wise
 if self.ndim == 1:
 raise ValueError(
 "Series.replace cannot use dict-value and "
 "non-None to_replace"
)
 mapping = {col: (to_replace, val) for col, val in value.items()}
 return self._replace_columnwise(mapping, inplace, regex)

 elif not is_list_like(value): # NA -> 0
 new_data = self._mgr.replace(
 to_replace=to_replace, value=value, inplace=inplace, regex=regex
)
 else:
 raise TypeError(
 f'Invalid "to_replace" type: {repr(type(to_replace).__name__)}'
)

result = self._constructor(new_data)
if inplace:
 return self._update_inplace(result)
else:
 return result.__finalize__(self, method="replace")

_shared_docs[
 "interpolate"
] = """
Please note that only ``method='linear'`` is supported for
DataFrame/Series with a MultiIndex.

Parameters

method : str, default 'linear'
 Interpolation technique to use. One of:

 * 'linear': Ignore the index and treat the values as equally
 spaced. This is the only method supported on MultiIndexes.
 * 'time': Works on daily and higher resolution data to interpolate
 given length of interval.
 * 'index', 'values': use the actual numerical values of the index.
 * 'pad': Fill in NaNs using existing values.
 * 'nearest', 'zero', 'slinear', 'quadratic', 'cubic', 'spline',
 'barycentric', 'polynomial': Passed to
 `scipy.interpolate.interp1d`. These methods use the numerical
 values of the index. Both 'polynomial' and 'spline' require that
"""

```

```
 you also specify an `order` (int), e.g.
 ``df.interpolate(method='polynomial', order=5)``.
* 'krogh', 'piecewise_polynomial', 'spline', 'pchip', 'akima',
 'cubicspline': Wrappers around the SciPy interpolation methods of
 similar names. See 'Notes'.
* 'from_derivatives': Refers to
 `scipy.interpolate.BPoly.from_derivatives` which
 replaces 'piecewise_polynomial' interpolation method in
 scipy 0.18.
axis : {0 or 'index', 1 or 'columns', None}, default None
 Axis to interpolate along.
limit : int, optional
 Maximum number of consecutive NaNs to fill. Must be greater than
 0.
inplace : bool, default False
 Update the data in place if possible.
limit_direction : {'forward', 'backward', 'both'}, default 'forward'
 If limit is specified, consecutive NaNs will be filled in this
 direction.
limit_area : {'None', 'inside', 'outside'}, default None
 If limit is specified, consecutive NaNs will be filled with this
 restriction.

* ``None``: No fill restriction.
* 'inside': Only fill NaNs surrounded by valid values
 (interpolate).
* 'outside': Only fill NaNs outside valid values (extrapolate).

.. versionadded:: 0.23.0

downcast : optional, 'infer' or None, defaults to None
 Downcast dtypes if possible.
**kwargs
 Keyword arguments to pass on to the interpolating function.

>Returns

Series or DataFrame
 Returns the same object type as the caller, interpolated at
 some or all ``NaN`` values.

See Also

fillna : Fill missing values using different methods.
scipy.interpolate.Akima1DInterpolator : Piecewise cubic polynomials
 (Akima interpolator).
scipy.interpolate.BPoly.from_derivatives : Piecewise polynomial in the
 Bernstein basis.
scipy.interpolate.interp1d : Interpolate a 1-D function.
scipy.interpolate.KroghInterpolator : Interpolate polynomial (Krogh
 interpolator).
scipy.interpolate.PchipInterpolator : PCHIP 1-d monotonic cubic
 interpolation.
scipy.interpolate.CubicSpline : Cubic spline data interpolator.

>Notes

The 'krogh', 'piecewise_polynomial', 'spline', 'pchip' and 'akima'
methods are wrappers around the respective SciPy implementations of
similar names. These use the actual numerical values of the index.
For more information on their behavior, see the
`SciPy documentation`
```

```

<https://docs.scipy.org/doc/scipy/reference/interpolate.html#univariate-interpolation>_
and `SciPy tutorial
<https://docs.scipy.org/doc/scipy/reference/tutorial/interpolate.html>`__.

Examples

Filling in ``NaN`` in a :class:`~pandas.Series` via linear interpolation.

>>> s = pd.Series([0, 1, np.nan, 3])
>>> s
0 0.0
1 1.0
2 NaN
3 3.0
dtype: float64
>>> s.interpolate()
0 0.0
1 1.0
2 2.0
3 3.0
dtype: float64

Filling in ``NaN`` in a Series by padding, but filling at most two consecutive ``NaN`` at a time.

>>> s = pd.Series([np.nan, "single_one", np.nan,
... "fill_two_more", np.nan, np.nan, np.nan,
... 4.71, np.nan])
>>> s
0 NaN
1 single_one
2 NaN
3 fill_two_more
4 NaN
5 NaN
6 NaN
7 4.71
8 NaN
dtype: object
>>> s.interpolate(method='pad', limit=2)
0 NaN
1 single_one
2 single_one
3 fill_two_more
4 fill_two_more
5 fill_two_more
6 NaN
7 4.71
8 4.71
dtype: object

Filling in ``NaN`` in a Series via polynomial interpolation or splines:
Both 'polynomial' and 'spline' methods require that you also specify an ``order`` (int).

>>> s = pd.Series([0, 2, np.nan, 8])
>>> s.interpolate(method='polynomial', order=2)
0 0.000000
1 2.000000
2 4.666667
3 8.000000

```

```
dtype: float64
```

Fill the DataFrame forward (that is, going down) along each column using linear interpolation.

Note how the last entry in column 'a' is interpolated differently, because there is no entry after it to use for interpolation.

Note how the first entry in column 'b' remains ``NaN``, because there is no entry before it to use for interpolation.

```
>>> df = pd.DataFrame([(0.0, np.nan, -1.0, 1.0),
... (np.nan, 2.0, np.nan, np.nan),
... (2.0, 3.0, np.nan, 9.0),
... (np.nan, 4.0, -4.0, 16.0)],
... columns=list('abcd'))
>>> df
 a b c d
0 0.0 NaN -1.0 1.0
1 NaN 2.0 NaN NaN
2 2.0 3.0 NaN 9.0
3 NaN 4.0 -4.0 16.0
>>> df.interpolate(method='linear', limit_direction='forward', axis=0)
 a b c d
0 0.0 NaN -1.0 1.0
1 1.0 2.0 -2.0 5.0
2 2.0 3.0 -3.0 9.0
3 2.0 4.0 -4.0 16.0
```

Using polynomial interpolation.

```
>>> df['d'].interpolate(method='polynomial', order=2)
0 1.0
1 4.0
2 9.0
3 16.0
Name: d, dtype: float64
"""
```

```
@Appender(_shared_docs["interpolate"] % _shared_doc_kwargs)
def interpolate(
 self,
 method="linear",
 axis=0,
 limit=None,
 inplace=False,
 limit_direction="forward",
 limit_area=None,
 downcast=None,
 **kwargs,
):
 """
 Interpolate values according to different methods.
 """
 inplace = validate_bool_kwarg(inplace, "inplace")
 axis = self._get_axis_number(axis)

 if axis == 0:
 df = self
 else:
 df = self.T

 if isinstance(df.index, MultiIndex) and method != "linear":
```

```

 raise ValueError(
 "Only `method=linear` interpolation is supported on MultiIndexes."
)

 if df.ndim == 2 and np.all(df.dtypes == np.dtype(object)):
 raise TypeError(
 "Cannot interpolate with all object-dtype columns "
 "in the DataFrame. Try setting at least one "
 "column to a numeric dtype."
)

create/use the index
if method == "linear":
 # prior default
 index = np.arange(len(df.index))
else:
 index = df.index
methods = {"index", "values", "nearest", "time"}
is_numeric_or_datetime = (
 is_numeric_dtype(index)
 or is_datetime64_any_dtype(index)
 or is_timedelta64_dtype(index)
)
if method not in methods and not is_numeric_or_datetime:
 raise ValueError(
 "Index column must be numeric or datetime type when "
 f"using {method} method other than linear."
 "Try setting a numeric or datetime index column before "
 "interpolating."
)

if isna(index).any():
 raise NotImplementedError(
 "Interpolation with NaNs in the index "
 "has not been implemented. Try filling "
 "those NaNs before interpolating."
)
data = df._mgr
new_data = data.interpolate(
 method=method,
 axis=self._info_axis_number,
 index=index,
 limit=limit,
 limit_direction=limit_direction,
 limit_area=limit_area,
 inplace=inplace,
 downcast=downcast,
 **kwargs,
)
result = self._constructor(new_data)
if axis == 1:
 result = result.T
if inplace:
 return self._update_inplace(result)
else:
 return result.__finalize__(self, method="interpolate")

Timeseries methods Methods

def asof(self, where, subset=None):
 """

```

```
Return the last row(s) without any NaNs before `where`.
```

The last row (for each element in `where`, if list) without any NaN is taken.

In case of a :class:`~pandas.DataFrame`, the last row without NaN considering only the subset of columns (if not `None`)

If there is no good value, NaN is returned for a Series or a Series of NaN values for a DataFrame

#### Parameters

-----

where : date or array-like of dates

Date(s) before which the last row(s) are returned.

subset : str or array-like of str, default `None`

For DataFrame, if not `None`, only use these columns to check for NaNs.

#### Returns

-----

scalar, Series, or DataFrame

The return can be:

- \* scalar : when `self` is a Series and `where` is a scalar
- \* Series: when `self` is a Series and `where` is an array-like, or when `self` is a DataFrame and `where` is a scalar
- \* DataFrame : when `self` is a DataFrame and `where` is an array-like

Return scalar, Series, or DataFrame.

#### See Also

-----

`merge_asof` : Perform an asof merge. Similar to left join.

#### Notes

-----

Dates are assumed to be sorted. Raises if this is not the case.

#### Examples

-----

A Series and a scalar `where`.

```
>>> s = pd.Series([1, 2, np.nan, 4], index=[10, 20, 30, 40])
>>> s
10 1.0
20 2.0
30 NaN
40 4.0
dtype: float64
```

```
>>> s.asof(20)
2.0
```

For a sequence `where`, a Series is returned. The first value is NaN, because the first element of `where` is before the first index value.

```
>>> s.asof([5, 20])
5 NaN
20 2.0
dtype: float64
```

```
Missing values are not considered. The following is ``2.0``, not
NaN, even though NaN is at the index location for ``30``.
```

```
>>> s.asof(30)
2.0
```

Take all columns into consideration

```
>>> df = pd.DataFrame({'a': [10, 20, 30, 40, 50],
... 'b': [None, None, None, None, 500]},
... index=pd.DatetimeIndex(['2018-02-27 09:01:00',
... '2018-02-27 09:02:00',
... '2018-02-27 09:03:00',
... '2018-02-27 09:04:00',
... '2018-02-27 09:05:00']))
>>> df.asof(pd.DatetimeIndex(['2018-02-27 09:03:30',
... '2018-02-27 09:04:30']))
 a b
2018-02-27 09:03:30 NaN NaN
2018-02-27 09:04:30 NaN NaN
```

Take a single column into consideration

```
>>> df.asof(pd.DatetimeIndex(['2018-02-27 09:03:30',
... '2018-02-27 09:04:30']),
... subset=['a'])
 a b
2018-02-27 09:03:30 30.0 NaN
2018-02-27 09:04:30 40.0 NaN
"""
if isinstance(where, str):
 where = Timestamp(where)

if not self.index.is_monotonic:
 raise ValueError("asof requires a sorted index")

is_series = isinstance(self, ABCSeries)
if is_series:
 if subset is not None:
 raise ValueError("subset is not valid for Series")
else:
 if subset is None:
 subset = self.columns
 if not is_list_like(subset):
 subset = [subset]

is_list = is_list_like(where)
if not is_list:
 start = self.index[0]
 if isinstance(self.index, PeriodIndex):
 where = Period(where, freq=self.index.freq)

 if where < start:
 if not is_series:
 from pandas import Series

 return Series(index=self.columns, name=where, dtype=np.float64)
 return np.nan

It's always much faster to use a *while* loop here for
Series than pre-computing all the NAs. However a
while loop is extremely expensive for DataFrame
```

```

so we later pre-compute all the NAs and use the same
code path whether *where* is a scalar or list.
See PR: https://github.com/pandas-dev/pandas/pull/14476
if is_series:
 loc = self.index.searchsorted(where, side="right")
 if loc > 0:
 loc -= 1

 values = self._values
 while loc > 0 and isna(values[loc]):
 loc -= 1
 return values[loc]

if not isinstance(where, Index):
 where = Index(where) if is_list else Index([where])

nulls = self.isna() if is_series else self[subset].isna().any(1)
if nulls.all():
 if is_series:
 return self._constructor(np.nan, index=where, name=self.name)
 elif is_list:
 from pandas import DataFrame

 return DataFrame(np.nan, index=where, columns=self.columns)
else:
 from pandas import Series

 return Series(np.nan, index=self.columns, name=where[0])

locs = self.index.asof_locs(where, ~(nulls._values))

mask the missing
missing = locs == -1
data = self.take(locs)
data.index = where
data.loc[missing] = np.nan
return data if is_list else data.iloc[-1]

Action Methods

_shared_docs[
 "isna"
] = """
Detect missing values.

Return a boolean same-sized object indicating if the values are NA.
NA values, such as None or :attr:`numpy.NaN`, gets mapped to True
values.
Everything else gets mapped to False values. Characters such as empty
strings ``''`` or :attr:`numpy.inf` are not considered NA values
(unless you set ``pandas.options.mode.use_inf_as_na = True``).

>Returns

%(klass)s
 Mask of bool values for each element in %(klass)s that
 indicates whether an element is not an NA value.

>See Also

%(klass)s.isnull : Alias of isna.
%(klass)s.notna : Boolean inverse of isna.

```

```

%(klass)s.dropna : Omit axes labels with missing values.
isna : Top-level isna.

Examples

Show which entries in a DataFrame are NA.

>>> df = pd.DataFrame({'age': [5, 6, np.nan],
... 'born': [pd.NaT, pd.Timestamp('1939-05-27'),
... pd.Timestamp('1940-04-25')],
... 'name': ['Alfred', 'Batman', ''],
... 'toy': [None, 'Batmobile', 'Joker']})
>>> df
 age born name toy
0 5.0 NaT Alfred None
1 6.0 1939-05-27 Batman Batmobile
2 NaN 1940-04-25 Joker

>>> df.isna()
 age born name toy
0 False True False True
1 False False False False
2 True False False False

Show which entries in a Series are NA.

>>> ser = pd.Series([5, 6, np.nan])
>>> ser
0 5.0
1 6.0
2 NaN
dtype: float64

>>> ser.isna()
0 False
1 False
2 True
dtype: bool
"""

@Appender(_shared_docs["isna"] % _shared_doc_kwargs)
def isna(self: FrameOrSeries) -> FrameOrSeries:
 return isna(self).__finalize__(self, method="isna")

@Appender(_shared_docs["isna"] % _shared_doc_kwargs)
def isnull(self: FrameOrSeries) -> FrameOrSeries:
 return isna(self).__finalize__(self, method="isnull")

_shared_docs[
 "notna"
] = """
Detect existing (non-missing) values.

Return a boolean same-sized object indicating if the values are not NA.
Non-missing values get mapped to True. Characters such as empty
strings ``''`` or :attr:`numpy.inf` are not considered NA values
(unless you set ``pandas.options.mode.use_inf_as_na = True``).
NA values, such as None or :attr:`numpy.NaN`, get mapped to False
values.

Returns

%(klass)s

```

```
Mask of bool values for each element in %(klass)s that
indicates whether an element is not an NA value.
```

See Also

```

%(klass)s.notnull : Alias of notna.
%(klass)s.isna : Boolean inverse of notna.
%(klass)s.dropna : Omit axes labels with missing values.
notna : Top-level notna.
```

Examples

```

Show which entries in a DataFrame are not NA.
```

```
>>> df = pd.DataFrame({'age': [5, 6, np.nan],
... 'born': [pd.NaT, pd.Timestamp('1939-05-27'),
... pd.Timestamp('1940-04-25')],
... 'name': ['Alfred', 'Batman', ''],
... 'toy': [None, 'Batmobile', 'Joker']})
>>> df
 age born name toy
0 5.0 NaT Alfred None
1 6.0 1939-05-27 Batman Batmobile
2 NaN 1940-04-25 ''

>>> df.notna()
 age born name toy
0 True False True False
1 True True True True
2 False True True True
```

```
Show which entries in a Series are not NA.
```

```
>>> ser = pd.Series([5, 6, np.nan])
>>> ser
0 5.0
1 6.0
2 NaN
dtype: float64

>>> ser.notna()
0 True
1 True
2 False
dtype: bool
"""
```

```
@Appender(_shared_docs["notna"] % _shared_doc_kwargs)
def notna(self: FrameOrSeries) -> FrameOrSeries:
 return notna(self).__finalize__(self, method="notna")

@Appender(_shared_docs["notna"] % _shared_doc_kwargs)
def notnull(self: FrameOrSeries) -> FrameOrSeries:
 return notna(self).__finalize__(self, method="notnull")

def _clip_with_scalar(self, lower, upper, inplace: bool_t = False):
 if (lower is not None and np.any(isna(lower))) or (
 upper is not None and np.any(isna(upper))
):
 raise ValueError("Cannot use an NA value as a clip threshold")

 result = self
 mask = isna(self._values)
```

```

with np.errstate(all="ignore"):
 if upper is not None:
 subset = self.to_numpy() <= upper
 result = result.where(subset, upper, axis=None, inplace=False)
 if lower is not None:
 subset = self.to_numpy() >= lower
 result = result.where(subset, lower, axis=None, inplace=False)

 if np.any(mask):
 result[mask] = np.nan

 if inplace:
 return self._update_inplace(result)
 else:
 return result

def _clip_with_one_bound(self, threshold, method, axis, inplace):
 if axis is not None:
 axis = self._get_axis_number(axis)

 # method is self.le for upper bound and self.ge for lower bound
 if is_scalar(threshold) and is_number(threshold):
 if method.__name__ == "le":
 return self._clip_with_scalar(None, threshold, inplace=inplace)
 return self._clip_with_scalar(threshold, None, inplace=inplace)

 subset = method(threshold, axis=axis) | isna(self)

 # GH #15390
 # In order for where method to work, the threshold must
 # be transformed to NDFrame from other array like structure.
 if (not isinstance(threshold, ABCSeries)) and is_list_like(threshold):
 if isinstance(self, ABCSeries):
 threshold = self._constructor(threshold, index=self.index)
 else:
 threshold = _align_method_FRAME(self, threshold, axis, flex=None)[1]
 return self.where(subset, threshold, axis=axis, inplace=inplace)

def clip(
 self: FrameOrSeries,
 lower=None,
 upper=None,
 axis=None,
 inplace: bool_t = False,
 *args,
 **kwargs,
) -> FrameOrSeries:
 """
 Trim values at input threshold(s).

 Assigns values outside boundary to boundary values. Thresholds
 can be singular values or array like, and in the latter case
 the clipping is performed element-wise in the specified axis.

 Parameters

 lower : float or array_like, default None
 Minimum threshold value. All values below this
 threshold will be set to it.
 upper : float or array_like, default None
 Maximum threshold value. All values above this
 """

```

```
 threshold will be set to it.
axis : int or str axis name, optional
 Align object with lower and upper along the given axis.
inplace : bool, default False
 Whether to perform the operation in place on the data.
*args, **kwargs
 Additional keywords have no effect but might be accepted
 for compatibility with numpy.
```

Returns

-----

Series or DataFrame

```
 Same type as calling object with the values outside the
 clip boundaries replaced.
```

See Also

-----

```
Series.clip : Trim values at input threshold in series.
DataFrame.clip : Trim values at input threshold in dataframe.
numpy.clip : Clip (limit) the values in an array.
```

Examples

-----

```
>>> data = {'col_0': [9, -3, 0, -1, 5], 'col_1': [-2, -7, 6, 8, -5]}
>>> df = pd.DataFrame(data)
>>> df
 col_0 col_1
0 9 -2
1 -3 -7
2 0 6
3 -1 8
4 5 -5
```

Clips per column using lower and upper thresholds:

```
>>> df.clip(-4, 6)
 col_0 col_1
0 6 -2
1 -3 -4
2 0 6
3 -1 6
4 5 -4
```

Clips using specific lower and upper thresholds per column element:

```
>>> t = pd.Series([2, -4, -1, 6, 3])
```

```
>>> t
0 2
1 -4
2 -1
3 6
4 3
dtype: int64
```

```
>>> df.clip(t, t + 4, axis=0)
```

```
 col_0 col_1
0 6 -2
1 -3 -4
2 0 3
3 6 8
4 5 3
```

"""

```
inplace = validate_bool_kwarg(inplace, "inplace")
```

```

axis = nv.validate_clip_with_axis(axis, args, kwargs)
if axis is not None:
 axis = self._get_axis_number(axis)

GH 17276
numpy doesn't like NaN as a clip value
so ignore
GH 19992
numpy doesn't drop a list-like bound containing NaN
if not is_list_like(lower) and np.any(isna(lower)):
 lower = None
if not is_list_like(upper) and np.any(isna(upper)):
 upper = None

GH 2747 (arguments were reversed)
if lower is not None and upper is not None:
 if is_scalar(lower) and is_scalar(upper):
 lower, upper = min(lower, upper), max(lower, upper)

fast-path for scalars
if (lower is None or (is_scalar(lower) and is_number(lower))) and (
 upper is None or (is_scalar(upper) and is_number(upper))
):
 return self._clip_with_scalar(lower, upper, inplace=inplace)

result = self
if lower is not None:
 result = result._clip_with_one_bound(
 lower, method=self.ge, axis=axis, inplace=inplace
)
if upper is not None:
 if inplace:
 result = self
 result = result._clip_with_one_bound(
 upper, method=self.le, axis=axis, inplace=inplace
)

return result

_shared_docs[
 "groupby"
] = """
Group %(klass)s using a mapper or by a Series of columns.

A groupby operation involves some combination of splitting the
object, applying a function, and combining the results. This can be
used to group large amounts of data and compute operations on these
groups.

Parameters

by : mapping, function, label, or list of labels
 Used to determine the groups for the groupby.
 If ``by`` is a function, it's called on each value of the object's
 index. If a dict or Series is passed, the Series or dict VALUES
 will be used to determine the groups (the Series' values are first
 aligned; see ``.align()`` method). If an ndarray is passed, the
 values are used as-is determine the groups. A label or list of
 labels may be passed to group by the columns in ``self``. Notice
 that a tuple is interpreted as a (single) key.
axis : {0 or 'index', 1 or 'columns'}, default 0
 Split along rows (0) or columns (1).
"""

```

```
level : int, level name, or sequence of such, default None
 If the axis is a MultiIndex (hierarchical), group by a particular
 level or levels.
as_index : bool, default True
 For aggregated output, return object with group labels as the
 index. Only relevant for DataFrame input. as_index=False is
 effectively "SQL-style" grouped output.
sort : bool, default True
 Sort group keys. Get better performance by turning this off.
 Note this does not influence the order of observations within each
 group. Groupby preserves the order of rows within each group.
group_keys : bool, default True
 When calling apply, add group keys to index to identify pieces.
squeeze : bool, default False
 Reduce the dimensionality of the return type if possible,
 otherwise return a consistent type.
observed : bool, default False
 This only applies if any of the groupers are Categoricals.
 If True: only show observed values for categorical groupers.
 If False: show all values for categorical groupers.

 .. versionadded:: 0.23.0

>Returns

%(klass)sGroupBy
 Returns a groupby object that contains information about the groups.

See Also

resample : Convenience method for frequency conversion and resampling
 of time series.

Notes

See the `user guide
<https://pandas.pydata.org/pandas-docs/stable/groupby.html>`_ for more.
"""

def asfreq(
 self: FrameOrSeries,
 freq,
 method=None,
 how: Optional[str] = None,
 normalize: bool_t = False,
 fill_value=None,
) -> FrameOrSeries:
 """
 Convert TimeSeries to specified frequency.

 Optionally provide filling method to pad/backfill missing values.

 Returns the original data conformed to a new index with the specified
 frequency. ``resample`` is more appropriate if an operation, such as
 summarization, is necessary to represent the data at the new frequency.

Parameters

freq : DateOffset or str
 Frequency DateOffset or string.
method : {'backfill'/'bfill', 'pad'/'ffill'}, default None
 Method to use for filling holes in reindexed Series (note this
 does not fill NaNs that already were present):

```

```
* 'pad' / 'ffill': propagate last valid observation forward to next
 valid
* 'backfill' / 'bfill': use NEXT valid observation to fill.
how : {'start', 'end'}, default end
 For PeriodIndex only (see PeriodIndex.asfreq).
normalize : bool, default False
 Whether to reset output index to midnight.
fill_value : scalar, optional
 Value to use for missing values, applied during upsampling (note
 this does not fill NaNs that already were present).

>Returns

Same type as caller
 Object converted to the specified frequency.

See Also

reindex : Conform DataFrame to new index with optional filling logic.

Notes

To learn more about the frequency strings, please see `this link
<https://pandas.pydata.org/pandas-docs/stable/user_guide/timeseries.html#offset-aliases>`__.

Examples

Start by creating a series with 4 one minute timestamps.

>>> index = pd.date_range('1/1/2000', periods=4, freq='T')
>>> series = pd.Series([0.0, None, 2.0, 3.0], index=index)
>>> df = pd.DataFrame({'s':series})
>>> df
 s
2000-01-01 00:00:00 0.0
2000-01-01 00:01:00 NaN
2000-01-01 00:02:00 2.0
2000-01-01 00:03:00 3.0

Upsample the series into 30 second bins.

>>> df.asfreq(freq='30S')
 s
2000-01-01 00:00:00 0.0
2000-01-01 00:00:30 NaN
2000-01-01 00:01:00 NaN
2000-01-01 00:01:30 NaN
2000-01-01 00:02:00 2.0
2000-01-01 00:02:30 NaN
2000-01-01 00:03:00 3.0

Upsample again, providing a ``fill value``.

>>> df.asfreq(freq='30S', fill_value=9.0)
 s
2000-01-01 00:00:00 0.0
2000-01-01 00:00:30 9.0
2000-01-01 00:01:00 NaN
2000-01-01 00:01:30 9.0
2000-01-01 00:02:00 2.0
2000-01-01 00:02:30 9.0
```

```
2000-01-01 00:03:00 3.0

Upsample again, providing a ``method``.

>>> df.asfreq(freq='30S', method='bfill')
 s
2000-01-01 00:00:00 0.0
2000-01-01 00:00:30 NaN
2000-01-01 00:01:00 NaN
2000-01-01 00:01:30 2.0
2000-01-01 00:02:00 2.0
2000-01-01 00:02:30 3.0
2000-01-01 00:03:00 3.0
"""

from pandas.core.resample import asfreq

return asfreq(
 self,
 freq,
 method=method,
 how=how,
 normalize=normalize,
 fill_value=fill_value,
)

def at_time(
 self: FrameOrSeries, time, asof: bool_t = False, axis=None
) -> FrameOrSeries:
"""
Select values at particular time of day (e.g., 9:30AM).

Parameters

time : datetime.time or str
axis : {0 or 'index', 1 or 'columns'}, default 0

.. versionadded:: 0.24.0

Returns

Series or DataFrame

Raises

TypeError
 If the index is not a :class:`DatetimeIndex`

See Also

between_time : Select values between particular times of the day.
first : Select initial periods of time series based on a date offset.
last : Select final periods of time series based on a date offset.
DatetimeIndex.indexer_at_time : Get just the index locations for
 values at particular time of the day.

Examples

>>> i = pd.date_range('2018-04-09', periods=4, freq='12H')
>>> ts = pd.DataFrame({'A': [1, 2, 3, 4]}, index=i)
>>> ts
 A
2018-04-09 00:00:00 1
2018-04-09 12:00:00 2
```

```
2018-04-10 00:00:00 3
2018-04-10 12:00:00 4

>>> ts.at_time('12:00')
 A
2018-04-09 12:00:00 2
2018-04-10 12:00:00 4
"""

if axis is None:
 axis = self._stat_axis_number
axis = self._get_axis_number(axis)

index = self._get_axis(axis)

if not isinstance(index, DatetimeIndex):
 raise TypeError("Index must be DatetimeIndex")

indexer = index.indexer_at_time(time, asof=asof)
return self._take_with_is_copy(indexer, axis=axis)

def between_time(
 self: FrameOrSeries,
 start_time,
 end_time,
 include_start: bool_t = True,
 include_end: bool_t = True,
 axis=None,
) -> FrameOrSeries:
"""
Select values between particular times of the day (e.g., 9:00-9:30 AM).

By setting ``start_time`` to be later than ``end_time``, you can get the times that are *not* between the two times.

Parameters

start_time : datetime.time or str
 Initial time as a time filter limit.
end_time : datetime.time or str
 End time as a time filter limit.
include_start : bool, default True
 Whether the start time needs to be included in the result.
include_end : bool, default True
 Whether the end time needs to be included in the result.
axis : {0 or 'index', 1 or 'columns'}, default 0
 Determine range time on index or columns value.

 .. versionadded:: 0.24.0

Returns

Series or DataFrame
 Data from the original object filtered to the specified dates range.

Raises

TypeError
 If the index is not a :class:`DatetimeIndex`

See Also

at_time : Select values at a particular time of the day.
first : Select initial periods of time series based on a date offset.
```

```
last : Select final periods of time series based on a date offset.
DatetimeIndex.indexer_between_time : Get just the index locations for
values between particular times of the day.
```

## Examples

-----

```
>>> i = pd.date_range('2018-04-09', periods=4, freq='1D20min')
>>> ts = pd.DataFrame({'A': [1, 2, 3, 4]}, index=i)
>>> ts
```

```
 A
2018-04-09 00:00:00 1
2018-04-10 00:20:00 2
2018-04-11 00:40:00 3
2018-04-12 01:00:00 4
```

```
>>> ts.between_time('0:15', '0:45')
```

```
 A
2018-04-10 00:20:00 2
2018-04-11 00:40:00 3
```

You get the times that are \*not\* between two times by setting  
` `start\_time` ` later than ``end\_time`` :

```
>>> ts.between_time('0:45', '0:15')
```

```
 A
2018-04-09 00:00:00 1
2018-04-12 01:00:00 4
"""
```

```
if axis is None:
 axis = self._stat_axis_number
axis = self._get_axis_number(axis)
```

```
index = self._get_axis(axis)
```

```
if not isinstance(index, DatetimeIndex):
 raise TypeError("Index must be DatetimeIndex")
```

```
indexer = index.indexer_between_time(
 start_time, end_time, include_start=include_start, include_end=include_end,
)
return self._take_with_is_copy(indexer, axis=axis)
```

```
def resample(
```

```
 self,
 rule,
 axis=0,
 closed: Optional[str] = None,
 label: Optional[str] = None,
 convention: str = "start",
 kind: Optional[str] = None,
 loffset=None,
 base: int = 0,
 on=None,
 level=None,
) -> "Resampler":
 """
```

```
 Resample time-series data.
```

Convenience method for frequency conversion and resampling of time series. Object must have a datetime-like index (`DatetimeIndex`, `PeriodIndex`, or `TimedeltaIndex`), or pass datetime-like values to the `on` or `level` keyword.

## Parameters

```

rule : DateOffset, Timedelta or str
 The offset string or object representing target conversion.
axis : {0 or 'index', 1 or 'columns'}, default 0
 Which axis to use for up- or down-sampling. For `Series` this
 will default to 0, i.e. along the rows. Must be
 `DatetimeIndex`, `TimedeltaIndex` or `PeriodIndex`.
closed : {'right', 'left'}, default None
 Which side of bin interval is closed. The default is 'left'
 for all frequency offsets except for 'M', 'A', 'Q', 'BM',
 'BA', 'BQ', and 'W' which all have a default of 'right'.
label : {'right', 'left'}, default None
 Which bin edge label to label bucket with. The default is 'left'
 for all frequency offsets except for 'M', 'A', 'Q', 'BM',
 'BA', 'BQ', and 'W' which all have a default of 'right'.
convention : {'start', 'end', 's', 'e'}, default 'start'
 For `PeriodIndex` only, controls whether to use the start or
 end of `rule`.
kind : {'timestamp', 'period'}, optional, default None
 Pass 'timestamp' to convert the resulting index to a
 `DateTimeIndex` or 'period' to convert it to a `PeriodIndex`.
 By default the input representation is retained.
loffset : timedelta, default None
 Adjust the resampled time labels.
base : int, default 0
 For frequencies that evenly subdivide 1 day, the "origin" of the
 aggregated intervals. For example, for '5min' frequency, base could
 range from 0 through 4. Defaults to 0.
on : str, optional
 For a DataFrame, column to use instead of index for resampling.
 Column must be datetime-like.

level : str or int, optional
 For a MultiIndex, level (name or number) to use for
 resampling. `level` must be datetime-like.
```

Returns

-----  
Resampler object

See Also

-----  
groupby : Group by mapping, function, label, or list of labels.  
Series.resample : Resample a Series.  
DataFrame.resample: Resample a DataFrame.

Notes

-----

See the `user guide`  
[<https://pandas.pydata.org/pandas-docs/stable/user\\_guide/timeseries.html#resampling>](https://pandas.pydata.org/pandas-docs/stable/user_guide/timeseries.html#resampling)\_\_  
for more.

To learn more about the offset strings, please see `this link`  
[<https://pandas.pydata.org/pandas-docs/stable/user\\_guide/timeseries.html#dateoffset-objects>](https://pandas.pydata.org/pandas-docs/stable/user_guide/timeseries.html#dateoffset-objects)\_\_.

Examples

-----

Start by creating a series with 9 one minute timestamps.

```
>>> index = pd.date_range('1/1/2000', periods=9, freq='T')
>>> series = pd.Series(range(9), index=index)
```

```
>>> series
2000-01-01 00:00:00 0
2000-01-01 00:01:00 1
2000-01-01 00:02:00 2
2000-01-01 00:03:00 3
2000-01-01 00:04:00 4
2000-01-01 00:05:00 5
2000-01-01 00:06:00 6
2000-01-01 00:07:00 7
2000-01-01 00:08:00 8
Freq: T, dtype: int64
```

Downsample the series into 3 minute bins and sum the values of the timestamps falling into a bin.

```
>>> series.resample('3T').sum()
2000-01-01 00:00:00 3
2000-01-01 00:03:00 12
2000-01-01 00:06:00 21
Freq: 3T, dtype: int64
```

Downsample the series into 3 minute bins as above, but label each bin using the right edge instead of the left. Please note that the value in the bucket used as the label is not included in the bucket, which it labels. For example, in the original series the bucket ``2000-01-01 00:03:00`` contains the value 3, but the summed value in the resampled bucket with the label ``2000-01-01 00:03:00`` does not include 3 (if it did, the summed value would be 6, not 3). To include this value close the right side of the bin interval as illustrated in the example below this one.

```
>>> series.resample('3T', label='right').sum()
2000-01-01 00:03:00 3
2000-01-01 00:06:00 12
2000-01-01 00:09:00 21
Freq: 3T, dtype: int64
```

Downsample the series into 3 minute bins as above, but close the right side of the bin interval.

```
>>> series.resample('3T', label='right', closed='right').sum()
2000-01-01 00:00:00 0
2000-01-01 00:03:00 6
2000-01-01 00:06:00 15
2000-01-01 00:09:00 15
Freq: 3T, dtype: int64
```

Upsample the series into 30 second bins.

```
>>> series.resample('30S').asfreq()[0:5] # Select first 5 rows
2000-01-01 00:00:00 0.0
2000-01-01 00:00:30 NaN
2000-01-01 00:01:00 1.0
2000-01-01 00:01:30 NaN
2000-01-01 00:02:00 2.0
Freq: 30S, dtype: float64
```

Upsample the series into 30 second bins and fill the ``NaN`` values using the ``pad`` method.

```
>>> series.resample('30S').pad()[0:5]
2000-01-01 00:00:00 0
2000-01-01 00:00:30 0
```

```
2000-01-01 00:01:00 1
2000-01-01 00:01:30 1
2000-01-01 00:02:00 2
Freq: 30S, dtype: int64
```

Upsample the series into 30 second bins and fill the ``NaN`` values using the ``bfill`` method.

```
>>> series.resample('30S').bfill()[0:5]
2000-01-01 00:00:00 0
2000-01-01 00:00:30 1
2000-01-01 00:01:00 1
2000-01-01 00:01:30 2
2000-01-01 00:02:00 2
Freq: 30S, dtype: int64
```

Pass a custom function via ``apply``

```
>>> def custom_resampler(array_like):
... return np.sum(array_like) + 5
...
>>> series.resample('3T').apply(custom_resampler)
2000-01-01 00:00:00 8
2000-01-01 00:03:00 17
2000-01-01 00:06:00 26
Freq: 3T, dtype: int64
```

For a Series with a PeriodIndex, the keyword `convention` can be used to control whether to use the start or end of `rule`.

Resample a year by quarter using 'start' `convention`. Values are assigned to the first quarter of the period.

```
>>> s = pd.Series([1, 2], index=pd.period_range('2012-01-01',
... freq='A',
... periods=2))
>>> s
2012 1
2013 2
Freq: A-DEC, dtype: int64
>>> s.resample('Q', convention='start').asfreq()
2012Q1 1.0
2012Q2 NaN
2012Q3 NaN
2012Q4 NaN
2013Q1 2.0
2013Q2 NaN
2013Q3 NaN
2013Q4 NaN
Freq: Q-DEC, dtype: float64
```

Resample quarters by month using 'end' `convention`. Values are assigned to the last month of the period.

```
>>> q = pd.Series([1, 2, 3, 4], index=pd.period_range('2018-01-01',
... freq='Q',
... periods=4))
>>> q
2018Q1 1
2018Q2 2
2018Q3 3
2018Q4 4
Freq: Q-DEC, dtype: int64
```

```
>>> q.resample('M', convention='end').asfreq()
2018-03 1.0
2018-04 NaN
2018-05 NaN
2018-06 2.0
2018-07 NaN
2018-08 NaN
2018-09 3.0
2018-10 NaN
2018-11 NaN
2018-12 4.0
Freq: M, dtype: float64
```

For DataFrame objects, the keyword `on` can be used to specify the column instead of the index for resampling.

```
>>> d = dict({'price': [10, 11, 9, 13, 14, 18, 17, 19],
... 'volume': [50, 60, 40, 100, 50, 100, 40, 50]})
>>> df = pd.DataFrame(d)
>>> df['week_starting'] = pd.date_range('01/01/2018',
... periods=8,
... freq='W')
>>> df
 price volume week_starting
0 10 50 2018-01-07
1 11 60 2018-01-14
2 9 40 2018-01-21
3 13 100 2018-01-28
4 14 50 2018-02-04
5 18 100 2018-02-11
6 17 40 2018-02-18
7 19 50 2018-02-25
>>> df.resample('M', on='week_starting').mean()
 price volume
week_starting
2018-01-31 10.75 62.5
2018-02-28 17.00 60.0
```

For a DataFrame with MultiIndex, the keyword `level` can be used to specify on which level the resampling needs to take place.

```
>>> days = pd.date_range('1/1/2000', periods=4, freq='D')
>>> d2 = dict({'price': [10, 11, 9, 13, 14, 18, 17, 19],
... 'volume': [50, 60, 40, 100, 50, 100, 40, 50]})
>>> df2 = pd.DataFrame(d2,
... index=pd.MultiIndex.from_product([days,
... ['morning',
... 'afternoon']],
...))
>>> df2
 price volume
2000-01-01 morning 10 50
 afternoon 11 60
2000-01-02 morning 9 40
 afternoon 13 100
2000-01-03 morning 14 50
 afternoon 18 100
2000-01-04 morning 17 40
 afternoon 19 50
>>> df2.resample('D', level=0).sum()
 price volume
2000-01-01 21 110
2000-01-02 22 140
```

```
2000-01-03 32 150
2000-01-04 36 90
"""
from pandas.core.resample import get_resampler

axis = self._get_axis_number(axis)
return get_resampler(
 self,
 freq=rule,
 label=label,
 closed=closed,
 axis=axis,
 kind=kind,
 loffset=loffset,
 convention=convention,
 base=base,
 key=on,
 level=level,
)

def first(self: FrameOrSeries, offset) -> FrameOrSeries:
"""
Select initial periods of time series data based on a date offset.

When having a DataFrame with dates as index, this function can
select the first few rows based on a date offset.

Parameters

offset : str, DateOffset or dateutil.relativedelta
 The offset length of the data that will be selected. For instance,
 '1M' will display all the rows having their index within the first month.

Returns

Series or DataFrame
 A subset of the caller.

Raises

TypeError
 If the index is not a :class:`DatetimeIndex`.

See Also

last : Select final periods of time series based on a date offset.
at_time : Select values at a particular time of the day.
between_time : Select values between particular times of the day.

Examples

>>> i = pd.date_range('2018-04-09', periods=4, freq='2D')
>>> ts = pd.DataFrame({'A': [1, 2, 3, 4]}, index=i)
>>> ts
 A
2018-04-09 1
2018-04-11 2
2018-04-13 3
2018-04-15 4

Get the rows for the first 3 days:

>>> ts.first('3D')
```

```

A
2018-04-09 1
2018-04-11 2

Notice the data for 3 first calendar days were returned, not the first
3 days observed in the dataset, and therefore data for 2018-04-13 was
not returned.
"""
if not isinstance(self.index, DatetimeIndex):
 raise TypeError("'first' only supports a DatetimeIndex index")

if len(self.index) == 0:
 return self

offset = to_offset(offset)
end_date = end = self.index[0] + offset

Tick-like, e.g. 3 weeks
if isinstance(offset, Tick):
 if end_date in self.index:
 end = self.index.searchsorted(end_date, side="left")
 return self.iloc[:end]

return self.loc[:end]

def last(self: FrameOrSeries, offset) -> FrameOrSeries:
"""
Select final periods of time series data based on a date offset.

When having a DataFrame with dates as index, this function can
select the last few rows based on a date offset.

Parameters

offset : str, DateOffset, dateutil.relativedelta
 The offset length of the data that will be selected. For instance,
 '3D' will display all the rows having their index within the last 3 days.

Returns

Series or DataFrame
 A subset of the caller.

Raises

TypeError
 If the index is not a :class:`DatetimeIndex`

See Also

first : Select initial periods of time series based on a date offset.
at_time : Select values at a particular time of the day.
between_time : Select values between particular times of the day.

Examples

>>> i = pd.date_range('2018-04-09', periods=4, freq='2D')
>>> ts = pd.DataFrame({'A': [1, 2, 3, 4]}, index=i)
>>> ts
A
2018-04-09 1
2018-04-11 2
2018-04-13 3

```

```
2018-04-15 4
```

```
Get the rows for the last 3 days:
```

```
>>> ts.last('3D')
```

```
A
```

```
2018-04-13 3
```

```
2018-04-15 4
```

```
Notice the data for 3 last calendar days were returned, not the last
3 observed days in the dataset, and therefore data for 2018-04-11 was
not returned.
```

```
"""
```

```
if not isinstance(self.index, DatetimeIndex):
 raise TypeError("'last' only supports a DatetimeIndex index")
```

```
if len(self.index) == 0:
 return self
```

```
offset = to_offset(offset)
```

```
start_date = self.index[-1] - offset
start = self.index.searchsorted(start_date, side="right")
return self.iloc[start:]
```

```
def rank(
 self: FrameOrSeries,
 axis=0,
 method: str = "average",
 numeric_only: Optional[bool_t] = None,
 na_option: str = "keep",
 ascending: bool_t = True,
 pct: bool_t = False,
) -> FrameOrSeries:
```

```
"""
```

```
Compute numerical data ranks (1 through n) along axis.
```

```
By default, equal values are assigned a rank that is the average of the
ranks of those values.
```

```
Parameters
```

```

```

```
axis : {0 or 'index', 1 or 'columns'}, default 0
 Index to direct ranking.
```

```
method : {'average', 'min', 'max', 'first', 'dense'}, default 'average'
 How to rank the group of records that have the same value (i.e. ties):
```

- \* average: average rank of the group
- \* min: lowest rank in the group
- \* max: highest rank in the group
- \* first: ranks assigned in order they appear in the array
- \* dense: like 'min', but rank always increases by 1 between groups.

```
numeric_only : bool, optional
```

```
 For DataFrame objects, rank only numeric columns if set to True.
```

```
na_option : {'keep', 'top', 'bottom'}, default 'keep'
```

```
 How to rank NaN values:
```

- \* keep: assign NaN rank to NaN values
- \* top: assign smallest rank to NaN values if ascending
- \* bottom: assign highest rank to NaN values if ascending.

```
ascending : bool, default True
```

```
 Whether or not the elements should be ranked in ascending order.
pct : bool, default False
 Whether or not to display the returned rankings in percentile
 form.

Returns

same type as caller
 Return a Series or DataFrame with data ranks as values.

See Also

core.groupby.GroupBy.rank : Rank of values within each group.
```

#### Examples

```
>>> df = pd.DataFrame(data={'Animal': ['cat', 'penguin', 'dog',
... 'spider', 'snake'],
... 'Number_legs': [4, 2, 4, 8, np.nan]})

>>> df
 Animal Number_legs
0 cat 4.0
1 penguin 2.0
2 dog 4.0
3 spider 8.0
4 snake NaN
```

The following example shows how the method behaves with the above parameters:

- \* `default_rank`: this is the default behaviour obtained without using any parameter.
- \* `max_rank`: setting ```method = 'max'``` the records that have the same values are ranked using the highest rank (e.g.: since 'cat' and 'dog' are both in the 2nd and 3rd position, rank 3 is assigned.)
- \* `NA_bottom`: choosing ```na_option = 'bottom'```, if there are records with `NaN` values they are placed at the bottom of the ranking.
- \* `pct_rank`: when setting ```pct = True```, the ranking is expressed as percentile rank.

```
>>> df['default_rank'] = df['Number_legs'].rank()
>>> df['max_rank'] = df['Number_legs'].rank(method='max')
>>> df['NA_bottom'] = df['Number_legs'].rank(na_option='bottom')
>>> df['pct_rank'] = df['Number_legs'].rank(pct=True)
>>> df
 Animal Number_legs default_rank max_rank NA_bottom pct_rank
0 cat 4.0 2.5 3.0 2.5 0.625
1 penguin 2.0 1.0 1.0 1.0 0.250
2 dog 4.0 2.5 3.0 2.5 0.625
3 spider 8.0 4.0 4.0 4.0 1.000
4 snake NaN NaN NaN 5.0 NaN
"""
axis = self._get_axis_number(axis)

if na_option not in {"keep", "top", "bottom"}:
 msg = "na_option must be one of 'keep', 'top', or 'bottom'"
 raise ValueError(msg)

def ranker(data):
 ranks = algos.rank(
 data.values,
 axis=axis,
 method=method,
```

```

 ascending=ascending,
 na_option=na_option,
 pct=pct,
)
 ranks = self._constructor(ranks, **data._construct_axes_dict())
 return ranks.__finalize__(self, method="rank")

if numeric_only is None, and we can't get anything, we try with
numeric_only=True
if numeric_only is None:
 try:
 return ranker(self)
 except TypeError:
 numeric_only = True

if numeric_only:
 data = self._get_numeric_data()
else:
 data = self

return ranker(data)

@doc(**_shared_doc_kwargs)
def align(
 self,
 other,
 join="outer",
 axis=None,
 level=None,
 copy=True,
 fill_value=None,
 method=None,
 limit=None,
 fill_axis=0,
 broadcast_axis=None,
):
 """
 Align two objects on their axes with the specified join method.

 Join method is specified for each axis Index.

 Parameters

 other : DataFrame or Series
 join : {'outer', 'inner', 'left', 'right'}, default 'outer'
 axis : allowed axis of the other object, default None
 Align on index (0), columns (1), or both (None).
 level : int or level name, default None
 Broadcast across a level, matching Index values on the
 passed MultiIndex level.
 copy : bool, default True
 Always returns new objects. If copy=False and no reindexing is
 required then original objects are returned.
 fill_value : scalar, default np.NaN
 Value to use for missing values. Defaults to NaN, but can be any
 "compatible" value.
 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. Must be greater than 0 if not None.
fill_axis : {axes_single_arg}, default 0
 Filling axis, method and limit.
broadcast_axis : {axes_single_arg}, default None
 Broadcast values along this axis, if aligning two objects of
 different dimensions.

Returns

(left, right) : ({klass}, type of other)
 Aligned objects.
"""

method = missing.clean_fill_method(method)

if broadcast_axis == 1 and self.ndim != other.ndim:
 if isinstance(self, ABCSeries):
 # this means other is a DataFrame, and we need to broadcast
 # self
 cons = self._constructor_expanddim
 df = cons(
 {c: self for c in other.columns}, **other._construct_axes_dict()
)
 return df._align_frame(
 other,
 join=join,
 axis=axis,
 level=level,
 copy=copy,
 fill_value=fill_value,
 method=method,
 limit=limit,
 fill_axis=fill_axis,
)
 elif isinstance(other, ABCSeries):
 # this means self is a DataFrame, and we need to broadcast
 # other
 cons = other._constructor_expanddim
 df = cons(
 {c: other for c in self.columns}, **self._construct_axes_dict()
)
 return self._align_frame(
 df,
 join=join,
 axis=axis,
 level=level,
 copy=copy,
 fill_value=fill_value,
 method=method,
 limit=limit,
 fill_axis=fill_axis,
)

if axis is not None:
 axis = self._get_axis_number(axis)
if isinstance(other, ABCDataFrame):
 return self._align_frame(
 other,

```

```

 join=join,
 axis=axis,
 level=level,
 copy=copy,
 fill_value=fill_value,
 method=method,
 limit=limit,
 fill_axis=fill_axis,
)
 elif isinstance(other, ABCSeries):
 return self._align_series(
 other,
 join=join,
 axis=axis,
 level=level,
 copy=copy,
 fill_value=fill_value,
 method=method,
 limit=limit,
 fill_axis=fill_axis,
)
 else: # pragma: no cover
 raise TypeError(f"unsupported type: {type(other)}")

def _align_frame(
 self,
 other,
 join="outer",
 axis=None,
 level=None,
 copy: bool_t = True,
 fill_value=None,
 method=None,
 limit=None,
 fill_axis=0,
):
 # defaults
 join_index, join_columns = None, None
 ilidx, iridx = None, None
 clidx, cridx = None, None

 is_series = isinstance(self, ABCSeries)

 if axis is None or axis == 0:
 if not self.index.equals(other.index):
 join_index, ilidx, iridx = self.index.join(
 other.index, how=join, level=level, return_indexers=True
)

 if axis is None or axis == 1:
 if not is_series and not self.columns.equals(other.columns):
 join_columns, clidx, cridx = self.columns.join(
 other.columns, how=join, level=level, return_indexers=True
)

 if is_series:
 reindexers = {0: [join_index, ilidx]}
 else:
 reindexers = {0: [join_index, ilidx], 1: [join_columns, clidx]}

 left = self._reindex_with_indexers(
 reindexers, copy=copy, fill_value=fill_value, allow_dups=True
)

```

```

other must be always DataFrame
right = other._reindex_with_indexers(
 {0: [join_index, iridx], 1: [join_columns, cridx]},
 copy=copy,
 fill_value=fill_value,
 allow_dups=True,
)

if method is not None:
 _left = left.fillna(method=method, axis=fill_axis, limit=limit)
 assert _left is not None # needed for mypy
 left = _left
 right = right.fillna(method=method, axis=fill_axis, limit=limit)

if DatetimeIndex have different tz, convert to UTC
if is_datetime64tz_dtype(left.index):
 if left.index.tz != right.index.tz:
 if join_index is not None:
 left.index = join_index
 right.index = join_index

return (
 left.__finalize__(self),
 right.__finalize__(other),
)

```

```

def _align_series(
 self,
 other,
 join="outer",
 axis=None,
 level=None,
 copy: bool_t = True,
 fill_value=None,
 method=None,
 limit=None,
 fill_axis=0,
):
 is_series = isinstance(self, ABCSeries)

 # series/series compat, other must always be a Series
 if is_series:
 if axis:
 raise ValueError("cannot align series to a series other than axis 0")

 # equal
 if self.index.equals(other.index):
 join_index, lidx, ridx = None, None, None
 else:
 join_index, lidx, ridx = self.index.join(
 other.index, how=join, level=level, return_indexers=True
)

 left = self._reindex_indexer(join_index, lidx, copy)
 right = other._reindex_indexer(join_index, ridx, copy)

 else:
 # one has > 1 ndim
 fdata = self._mgr
 if axis == 0:
 join_index = self.index
 lidx, ridx = None, None

```

```

 if not self.index.equals(other.index):
 join_index, lidx, ridx = self.index.join(
 other.index, how=join, level=level, return_indexers=True
)

 if lidx is not None:
 fdata = fdata.reindex_indexer(join_index, lidx, axis=1)

 elif axis == 1:
 join_index = self.columns
 lidx, ridx = None, None
 if not self.columns.equals(other.index):
 join_index, lidx, ridx = self.columns.join(
 other.index, how=join, level=level, return_indexers=True
)

 if lidx is not None:
 fdata = fdata.reindex_indexer(join_index, lidx, axis=0)
 else:
 raise ValueError("Must specify axis=0 or 1")

 if copy and fdata is self._mgr:
 fdata = fdata.copy()

 left = self._constructor(fdata)

 if ridx is None:
 right = other
 else:
 right = other.reindex(join_index, level=level)

 # fill
 fill_na = notna(fill_value) or (method is not None)
 if fill_na:
 left = left.fillna(fill_value, method=method, limit=limit, axis=fill_axis)
 right = right.fillna(fill_value, method=method, limit=limit)

 # if DatetimeIndex have different tz, convert to UTC
 if is_series or (not is_series and axis == 0):
 if is_datetime64tz_dtype(left.index):
 if left.index.tz != right.index.tz:
 if join_index is not None:
 left.index = join_index
 right.index = join_index

 return (
 left.__finalize__(self),
 right.__finalize__(other),
)

def _where(
 self,
 cond,
 other=np.nan,
 inplace=False,
 axis=None,
 level=None,
 errors="raise",
 try_cast=False,
):
 """
 Equivalent to public method `where`, except that `other` is not
 applied as a function even if callable. Used in __setitem__.
 """

```

```

"""
inplace = validate_bool_kwarg(inplace, "inplace")

align the cond to same shape as myself
cond = com.apply_if_callable(cond, self)
if isinstance(cond, NDFrame):
 cond, _ = cond.align(self, join="right", broadcast_axis=1)
else:
 if not hasattr(cond, "shape"):
 cond = np.asarray(cond)
 if cond.shape != self.shape:
 raise ValueError("Array conditional must be same shape as self")
 cond = self._constructor(cond, **self._construct_axes_dict())

make sure we are boolean
fill_value = bool(inplace)
cond = cond.fillna(fill_value)

msg = "Boolean array expected for the condition, not {dtype}"

if not isinstance(cond, ABCDataFrame):
 # This is a single-dimensional object.
 if not is_bool_dtype(cond):
 raise ValueError(msg.format(dtype=cond.dtype))
elif not cond.empty:
 for dt in cond.dtypes:
 if not is_bool_dtype(dt):
 raise ValueError(msg.format(dtype=dt))
else:
 # GH#21947 we have an empty DataFrame, could be object-dtype
 cond = cond.astype(bool)

cond = -cond if inplace else cond

try to align with other
try_quick = True
if isinstance(other, NDFrame):

 # align with me
 if other.ndim <= self.ndim:

 _, other = self.align(
 other, join="left", axis=axis, level=level, fill_value=np.nan
)

 # if we are NOT aligned, raise as we cannot where index
 if axis is None and not all(
 other._get_axis(i).equals(ax) for i, ax in enumerate(self.axes)
):
 raise InvalidIndexError

 # slice me out of the other
 else:
 raise NotImplementedError(
 "cannot align with a higher dimensional NDFrame"
)

if isinstance(other, np.ndarray):

 if other.shape != self.shape:

 if self.ndim == 1:

```

```

 icond = cond._values

 # GH 2745 / GH 4192
 # treat like a scalar
 if len(other) == 1:
 other = other[0]

 # GH 3235
 # match True cond to other
 elif len(cond[icond]) == len(other):
 # try to not change dtype at first (if try_quick)
 if try_quick:
 new_other = np.asarray(self)
 new_other = new_other.copy()
 new_other[icond] = other
 other = new_other

 else:
 raise ValueError(
 "Length of replacements must equal series length"
)

 else:
 raise ValueError(
 "other must be the same shape as self when an ndarray"
)

 # we are the same shape, so create an actual object for alignment
 else:
 other = self._constructor(other, **self._construct_axes_dict())

if axis is None:
 axis = 0

if self.ndim == getattr(other, "ndim", 0):
 align = True
else:
 align = self._get_axis_number(axis) == 1

if align and isinstance(other, NDFrame):
 other = other.reindex(self._info_axis, axis=self._info_axis_number)
if isinstance(cond, NDFrame):
 cond = cond.reindex(self._info_axis, axis=self._info_axis_number)

block_axis = self._get_block_manager_axis(axis)

if inplace:
 # we may have different type blocks come out of putmask, so
 # reconstruct the block manager

 self._check_inplace_setting(other)
 new_data = self._mgr.putmask(
 mask=cond, new=other, align=align, axis=block_axis,
)
 result = self._constructor(new_data)
 return self._update_inplace(result)

else:
 new_data = self._mgr.where(
 other=other,
 cond=cond,
 align=align,

```

```

 errors=errors,
 try_cast=try_cast,
 axis=block_axis,
)
 result = self._constructor(new_data)
 return result.__finalize__(self)

/shared_docs[
 "where"
] = """
Replace values where the condition is %(cond_rev)s.

Parameters

cond : bool %(klass)s, array-like, or callable
 Where `cond` is %(cond)s, keep the original value. Where
 %(cond_rev)s, replace with corresponding value from `other`.
 If `cond` is callable, it is computed on the %(klass)s and
 should return boolean %(klass)s or array. The callable must
 not change input %(klass)s (though pandas doesn't check it).
other : scalar, %(klass)s, or callable
 Entries where `cond` is %(cond_rev)s are replaced with
 corresponding value from `other`.
 If other is callable, it is computed on the %(klass)s and
 should return scalar or %(klass)s. The callable must not
 change input %(klass)s (though pandas doesn't check it).
inplace : bool, default False
 Whether to perform the operation in place on the data.
axis : int, default None
 Alignment axis if needed.
level : int, default None
 Alignment level if needed.
errors : str, {'raise', 'ignore'}, default 'raise'
 Note that currently this parameter won't affect
 the results and will always coerce to a suitable dtype.

 - 'raise' : allow exceptions to be raised.
 - 'ignore' : suppress exceptions. On error return original object.

try_cast : bool, default False
 Try to cast the result back to the input type (if possible).

Returns

Same type as caller

See Also

:func:`DataFrame.%s` : Return an object of same shape as
 self.

Notes

The %(name)s method is an application of the if-then idiom. For each
element in the calling DataFrame, if ``cond`` is ``%(cond)s`` the
element is used; otherwise the corresponding element from the DataFrame
``other`` is used.

The signature for :func:`DataFrame.where` differs from
:func:`numpy.where`. Roughly ``df1.where(m, df2)`` is equivalent to
``np.where(m, df1, df2)``.

For further details and examples see the ``%(name)s`` documentation in

```

```
:ref:`indexing <indexing.where_mask>`.
```

## Examples

---

```
>>> s = pd.Series(range(5))
>>> s.where(s > 0)
0 NaN
1 1.0
2 2.0
3 3.0
4 4.0
dtype: float64

>>> s.mask(s > 0)
0 0.0
1 NaN
2 NaN
3 NaN
4 NaN
dtype: float64

>>> s.where(s > 1, 10)
0 10
1 10
2 2
3 3
4 4
dtype: int64

>>> df = pd.DataFrame(np.arange(10).reshape(-1, 2), columns=['A', 'B'])
>>> df
 A B
0 0 1
1 2 3
2 4 5
3 6 7
4 8 9
>>> m = df % 3 == 0
>>> df.where(m, -df)
 A B
0 0 -1
1 -2 3
2 -4 -5
3 6 -7
4 -8 9
>>> df.where(m, -df) == np.where(m, df, -df)
 A B
0 True True
1 True True
2 True True
3 True True
4 True True
>>> df.where(m, -df) == df.mask(~m, -df)
 A B
0 True True
1 True True
2 True True
3 True True
4 True True
"""

```

```
@Appender(
 shared_docs["where"])
```

```

% dict(
 _shared_doc_kwargs,
 cond="True",
 cond_rev="False",
 name="where",
 name_other="mask",
)
)
def where(
 self,
 cond,
 other=np.nan,
 inplace=False,
 axis=None,
 level=None,
 errors="raise",
 try_cast=False,
):
 other = com.apply_if_callable(other, self)
 return self._where(
 cond, other, inplace, axis, level, errors=errors, try_cast=try_cast
)

@Appender(
 _shared_docs["where"]
)
% dict(
 _shared_doc_kwargs,
 cond="False",
 cond_rev="True",
 name="mask",
 name_other="where",
)
)
def mask(
 self,
 cond,
 other=np.nan,
 inplace=False,
 axis=None,
 level=None,
 errors="raise",
 try_cast=False,
):
 inplace = validate_bool_kwarg(inplace, "inplace")
 cond = com.apply_if_callable(cond, self)

 # see gh-21891
 if not hasattr(cond, "__invert__"):
 cond = np.array(cond)

 return self.where(
 ~cond,
 other=other,
 inplace=inplace,
 axis=axis,
 level=level,
 try_cast=try_cast,
 errors=errors,
)

@doc(klass=_shared_doc_kwargs["klass"])

```

```
def shift(
 self: FrameOrSeries, periods=1, freq=None, axis=0, fill_value=None
) -> FrameOrSeries:
 """
 Shift index by desired number of periods with an optional time `freq`.

 When `freq` is not passed, shift the index without realigning the data.
 If `freq` is passed (in this case, the index must be date or datetime,
 or it will raise a `NotImplementedError`), the index will be
 increased using the periods and the `freq`.

```

**Parameters**

-----

**periods** : int  
Number of periods to shift. Can be positive or negative.

**freq** : DateOffset, tseries.offsets, timedelta, or str, optional  
Offset to use from the tseries module or time rule (e.g. 'EOM').  
If `freq` is specified then the index values are shifted but the  
data is not realigned. That is, use `freq` if you would like to  
extend the index when shifting and preserve the original data.

**axis** : {{0 or 'index', 1 or 'columns', None}}, default None  
Shift direction.

**fill\_value** : object, optional  
The scalar value to use for newly introduced missing values.  
the default depends on the dtype of `self`.  
For numeric data, ``np.nan`` is used.  
For datetime, timedelta, or period data, etc. :attr:`NaT` is used.  
For extension dtypes, ``self.dtype.na\_value`` is used.

.. versionchanged:: 0.24.0

**Returns**

-----

{klass}  
Copy of input object, shifted.

**See Also**

-----

Index.shift : Shift values of Index.  
DatetimeIndex.shift : Shift values of DatetimeIndex.  
PeriodIndex.shift : Shift values of PeriodIndex.  
tshift : Shift the time index, using the index's frequency if  
available.

**Examples**

-----

```
>>> df = pd.DataFrame({{'Col1': [10, 20, 15, 30, 45],
... 'Col2': [13, 23, 18, 33, 48],
... 'Col3': [17, 27, 22, 37, 52]}})

>>> df.shift(periods=3)
 Col1 Col2 Col3
0 NaN NaN NaN
1 NaN NaN NaN
2 NaN NaN NaN
3 10.0 13.0 17.0
4 20.0 23.0 27.0

>>> df.shift(periods=1, axis='columns')
 Col1 Col2 Col3
0 NaN 10.0 13.0
1 NaN 20.0 23.0
2 NaN 15.0 18.0
```

```

3 NaN 30.0 33.0
4 NaN 45.0 48.0

>>> df.shift(periods=3, fill_value=0)
 Col1 Col2 Col3
0 0 0 0
1 0 0 0
2 0 0 0
3 10 13 17
4 20 23 27
"""
if periods == 0:
 return self.copy()

block_axis = self._get_block_manager_axis(axis)
if freq is None:
 new_data = self._mgr.shift(
 periods=periods, axis=block_axis, fill_value=fill_value
)
else:
 return self.tshift(periods, freq)

return self._constructor(new_data).__finalize__(self, method="shift")

def slice_shift(self: FrameOrSeries, periods: int = 1, axis=0) -> FrameOrSeries:
"""
Equivalent to `shift` without copying data.

The shifted data will not include the dropped periods and the
shifted axis will be smaller than the original.

Parameters

periods : int
 Number of periods to move, can be positive or negative.

Returns

shifted : same type as caller

Notes

While the `slice_shift` is faster than `shift`, you may pay for it
later during alignment.
"""
if periods == 0:
 return self

if periods > 0:
 vslicer = slice(None, -periods)
 islicer = slice(periods, None)
else:
 vslicer = slice(-periods, None)
 islicer = slice(None, periods)

new_obj = self._slice(vslider, axis=axis)
shifted_axis = self._get_axis(axis)[islicer]
new_obj.set_axis(shifted_axis, axis=axis, inplace=True)

return new_obj.__finalize__(self, method="slice_shift")

def tshift(
 self: FrameOrSeries, periods: int = 1, freq=None, axis: Axis = 0
):
 """
 Shift the index by a specified number of periods.

 Parameters

 periods : int
 Number of periods to move, can be positive or negative.
 freq : str or DateOffset, optional
 Frequency to use for the shift. If None, the period will be inferred
 from the data.
 axis : int, optional
 Axis along which to shift. Default is 0.
 """
 if freq is None:
 freq = infer_freq(self, periods)
 else:
 freq = to_offset(freq)
 if freq.n == 0:
 raise ValueError("freq must be non-zero")
 if freq.n < 0:
 periods = -periods
 if periods == 0:
 return self
 if periods > 0:
 vslicer = slice(None, -periods)
 islicer = slice(periods, None)
 else:
 vslicer = slice(-periods, None)
 islicer = slice(None, periods)

 new_obj = self._slice(vslider, axis=axis)
 shifted_axis = self._get_axis(axis)[islicer]
 new_obj.set_axis(shifted_axis, axis=axis, inplace=True)

 return new_obj.__finalize__(self, method="tshift")

```

```

) -> FrameOrSeries:
 """
 Shift the time index, using the index's frequency if available.

 Parameters

 periods : int
 Number of periods to move, can be positive or negative.
 freq : DateOffset, timedelta, or str, default None
 Increment to use from the tseries module
 or time rule expressed as a string (e.g. 'EOM').
 axis : {0 or 'index', 1 or 'columns', None}, default 0
 Corresponds to the axis that contains the Index.

 Returns

 shifted : Series/DataFrame

 Notes

 If freq is not specified then tries to use the freq or inferred_freq
 attributes of the index. If neither of those attributes exist, a
 ValueError is thrown
 """
 index = self._get_axis(axis)
 if freq is None:
 freq = getattr(index, "freq", None)

 if freq is None:
 freq = getattr(index, "inferred_freq", None)

 if freq is None:
 msg = "Freq was not given and was not set in the index"
 raise ValueError(msg)

 if periods == 0:
 return self

 if isinstance(freq, str):
 freq = to_offset(freq)

 axis = self._get_axis_number(axis)
 if isinstance(index, PeriodIndex):
 orig_freq = to_offset(index.freq)
 if freq != orig_freq:
 assert orig_freq is not None # for mypy
 raise ValueError(
 f"Given freq {freq.rule_code} does not match "
 f"PeriodIndex freq {orig_freq.rule_code}"
)
 new_ax = index.shift(periods)
 else:
 new_ax = index.shift(periods, freq)

 result = self.copy()
 result.set_axis(new_ax, axis, inplace=True)
 return result.__finalize__(self, method="tshift")

def truncate(
 self: FrameOrSeries, before=None, after=None, axis=None, copy: bool_t = True
) -> FrameOrSeries:
 """
 Truncate a Series or DataFrame before and after some index value.

```

This is a useful shorthand for boolean indexing based on index values above or below certain thresholds.

#### Parameters

-----

before : date, str, int  
Truncate all rows before this index value.  
after : date, str, int  
Truncate all rows after this index value.  
axis : {0 or 'index', 1 or 'columns'}, optional  
Axis to truncate. Truncates the index (rows) by default.  
copy : bool, default is True,  
Return a copy of the truncated section.

#### Returns

-----

type of caller  
The truncated Series or DataFrame.

#### See Also

-----

DataFrame.loc : Select a subset of a DataFrame by label.  
DataFrame.iloc : Select a subset of a DataFrame by position.

#### Notes

-----

If the index being truncated contains only datetime values, `before` and `after` may be specified as strings instead of Timestamps.

#### Examples

-----

```
>>> df = pd.DataFrame({'A': ['a', 'b', 'c', 'd', 'e'],
... 'B': ['f', 'g', 'h', 'i', 'j'],
... 'C': ['k', 'l', 'm', 'n', 'o']},
... index=[1, 2, 3, 4, 5])
>>> df
 A B C
1 a f k
2 b g l
3 c h m
4 d i n
5 e j o

>>> df.truncate(before=2, after=4)
 A B C
2 b g l
3 c h m
4 d i n
```

The columns of a DataFrame can be truncated.

```
>>> df.truncate(before="A", after="B", axis="columns")
 A B
1 a f
2 b g
3 c h
4 d i
5 e j
```

For Series, only rows can be truncated.

```
>>> df['A'].truncate(before=2, after=4)
2 b
3 c
4 d
Name: A, dtype: object
```

The index values in ``truncate`` can be datetimes or string dates.

```
>>> dates = pd.date_range('2016-01-01', '2016-02-01', freq='s')
>>> df = pd.DataFrame(index=dates, data={'A': 1})
>>> df.tail()
```

```
 A
2016-01-31 23:59:56 1
2016-01-31 23:59:57 1
2016-01-31 23:59:58 1
2016-01-31 23:59:59 1
2016-02-01 00:00:00 1
```

```
>>> df.truncate(before=pd.Timestamp('2016-01-05'),
... after=pd.Timestamp('2016-01-10')).tail()
```

```
 A
2016-01-09 23:59:56 1
2016-01-09 23:59:57 1
2016-01-09 23:59:58 1
2016-01-09 23:59:59 1
2016-01-10 00:00:00 1
```

Because the index is a DatetimeIndex containing only dates, we can specify `before` and `after` as strings. They will be coerced to Timestamps before truncation.

```
>>> df.truncate('2016-01-05', '2016-01-10').tail()
```

```
 A
2016-01-09 23:59:56 1
2016-01-09 23:59:57 1
2016-01-09 23:59:58 1
2016-01-09 23:59:59 1
2016-01-10 00:00:00 1
```

Note that ``truncate`` assumes a 0 value for any unspecified time component (midnight). This differs from partial string slicing, which returns any partially matching dates.

```
>>> df.loc['2016-01-05':'2016-01-10', :].tail()
```

```
 A
2016-01-10 23:59:55 1
2016-01-10 23:59:56 1
2016-01-10 23:59:57 1
2016-01-10 23:59:58 1
2016-01-10 23:59:59 1
"""
if axis is None:
 axis = self._stat_axis_number
axis = self._get_axis_number(axis)
ax = self._get_axis(axis)

GH 17935
Check that index is sorted
if not ax.is_monotonic_increasing and not ax.is_monotonic_decreasing:
 raise ValueError("truncate requires a sorted index")

if we have a date index, convert to dates, otherwise
```

```

treat like a slice
if ax.is_all_dates:
 from pandas.core.tools.datetimes import to_datetime

 before = to_datetime(before)
 after = to_datetime(after)

if before is not None and after is not None:
 if before > after:
 raise ValueError(f"Truncate: {after} must be after {before}")

slicer = [slice(None, None)] * self._AXIS_LEN
slicer[axis] = slice(before, after)
result = self.loc[tuple(slicer)]

if isinstance(ax, MultiIndex):
 setattr(result, self._get_axis_name(axis), ax.truncate(before, after))

if copy:
 result = result.copy()

return result

def tz_convert(
 self: FrameOrSeries, tz, axis=0, level=None, copy: bool_t = True
) -> FrameOrSeries:
 """
 Convert tz-aware axis to target time zone.

 Parameters

 tz : str or tzinfo object
 axis : the axis to convert
 level : int, str, default None
 If axis is a MultiIndex, convert a specific level. Otherwise
 must be None.
 copy : bool, default True
 Also make a copy of the underlying data.

 Returns

 %(klass)s
 Object with time zone converted axis.

 Raises

 TypeError
 If the axis is tz-naive.
 """
 axis = self._get_axis_number(axis)
 ax = self._get_axis(axis)

 def _tz_convert(ax, tz):
 if not hasattr(ax, "tz_convert"):
 if len(ax) > 0:
 ax_name = self._get_axis_name(axis)
 raise TypeError(
 f"{ax_name} is not a valid DatetimeIndex or PeriodIndex"
)
 else:
 ax = DatetimeIndex([], tz=tz)
 else:
 ax = ax.tz_convert(tz)

 return _tz_convert(result, tz)

```

```

 return ax

 # if a level is given it must be a MultiIndex level or
 # equivalent to the axis name
 if isinstance(ax, MultiIndex):
 level = ax._get_level_number(level)
 new_level = _tz_convert(ax.levels[level], tz)
 ax = ax.set_levels(new_level, level=level)
 else:
 if level not in (None, 0, ax.name):
 raise ValueError(f"The level {level} is not valid")
 ax = _tz_convert(ax, tz)

 result = self.copy(deep=copy)
 result = result.set_axis(ax, axis=axis, inplace=False)
 return result.__finalize__(self, method="tz_convert")

def tz_localize(
 self: FrameOrSeries,
 tz,
 axis=0,
 level=None,
 copy: bool_t = True,
 ambiguous="raise",
 nonexistent: str = "raise",
) -> FrameOrSeries:
 """
 Localize tz-naive index of a Series or DataFrame to target time zone.

 This operation localizes the Index. To localize the values in a
 timezone-naive Series, use :meth:`Series.dt.tz_localize`.

 Parameters

 tz : str or tzinfo
 axis : the axis to localize
 level : int, str, default None
 If axis is a MultiIndex, localize a specific level. Otherwise
 must be None.
 copy : bool, default True
 Also make a copy of the underlying data.
 ambiguous : 'infer', bool-ndarray, 'NaT', default 'raise'
 When clocks moved backward due to DST, ambiguous times may arise.
 For example in Central European Time (UTC+01), when going from
 03:00 DST to 02:00 non-DST, 02:30:00 local time occurs both at
 00:30:00 UTC and at 01:30:00 UTC. In such a situation, the
 `ambiguous` parameter dictates how ambiguous times should be
 handled.

 - 'infer' will attempt to infer fall dst-transition hours based on
 order
 - bool-ndarray where True signifies a DST time, False designates
 a non-DST time (note that this flag is only applicable for
 ambiguous times)
 - 'NaT' will return NaT where there are ambiguous times
 - 'raise' will raise an AmbiguousTimeError if there are ambiguous
 times.

 nonexistent : str, default 'raise'
 A nonexistent time does not exist in a particular timezone
 where clocks moved forward due to DST. Valid values are:

 - 'shift_forward' will shift the nonexistent time forward to the
 closest existing time

```

```

 - 'shift_backward' will shift the nonexistent time backward to the
 closest existing time
 - 'NaT' will return NaT where there are nonexistent times
 - timedelta objects will shift nonexistent times by the timedelta
 - 'raise' will raise an NonExistentTimeError if there are
 nonexistent times.

.. versionadded:: 0.24.0

>Returns

Series or DataFrame
 Same type as the input.

>Raises

TypeError
 If the TimeSeries is tz-aware and tz is not None.

>Examples

Localize local times:

>>> s = pd.Series([1],
... index=pd.DatetimeIndex(['2018-09-15 01:30:00']))
>>> s.tz_localize('CET')
2018-09-15 01:30:00+02:00 1
dtype: int64

Be careful with DST changes. When there is sequential data, pandas
can infer the DST time:

>>> s = pd.Series(range(7),
... index=pd.DatetimeIndex(['2018-10-28 01:30:00',
... '2018-10-28 02:00:00',
... '2018-10-28 02:30:00',
... '2018-10-28 02:00:00',
... '2018-10-28 02:30:00',
... '2018-10-28 03:00:00',
... '2018-10-28 03:30:00']))
>>> s.tz_localize('CET', ambiguous='infer')
2018-10-28 01:30:00+02:00 0
2018-10-28 02:00:00+02:00 1
2018-10-28 02:30:00+02:00 2
2018-10-28 02:00:00+01:00 3
2018-10-28 02:30:00+01:00 4
2018-10-28 03:00:00+01:00 5
2018-10-28 03:30:00+01:00 6
dtype: int64

In some cases, inferring the DST is impossible. In such cases, you can
pass an ndarray to the ambiguous parameter to set the DST explicitly

>>> s = pd.Series(range(3),
... index=pd.DatetimeIndex(['2018-10-28 01:20:00',
... '2018-10-28 02:36:00',
... '2018-10-28 03:46:00']))
>>> s.tz_localize('CET', ambiguous=np.array([True, True, False]))
2018-10-28 01:20:00+02:00 0
2018-10-28 02:36:00+02:00 1
2018-10-28 03:46:00+01:00 2
dtype: int64

```

```

If the DST transition causes nonexistent times, you can shift these
dates forward or backwards with a timedelta object or ``shift_forward``
or ``shift_backwards``.
>>> s = pd.Series(range(2),
... index=pd.DatetimeIndex(['2015-03-29 02:30:00',
... '2015-03-29 03:30:00']))
...
>>> s.tz_localize('Europe/Warsaw', nonexistent='shift_forward')
2015-03-29 03:00:00+02:00 0
2015-03-29 03:30:00+02:00 1
dtype: int64
>>> s.tz_localize('Europe/Warsaw', nonexistent='shift_backward')
2015-03-29 01:59:59.999999999+01:00 0
2015-03-29 03:30:00+02:00 1
dtype: int64
>>> s.tz_localize('Europe/Warsaw', nonexistent=pd.Timedelta('1H'))
2015-03-29 03:30:00+02:00 0
2015-03-29 03:30:00+02:00 1
dtype: int64
"""

nonexistent_options = ("raise", "NaT", "shift_forward", "shift_backward")
if nonexistent not in nonexistent_options and not isinstance(
 nonexistent, timedelta
):
 raise ValueError(
 "The nonexistent argument must be one of 'raise', "
 "'NaT', 'shift_forward', 'shift_backward' or "
 "a timedelta object"
)

axis = self._get_axis_number(axis)
ax = self._get_axis(axis)

def _tz_localize(ax, tz, ambiguous, nonexistent):
 if not hasattr(ax, "tz_localize"):
 if len(ax) > 0:
 ax_name = self._get_axis_name(axis)
 raise TypeError(
 f"{ax_name} is not a valid DatetimeIndex or PeriodIndex"
)
 else:
 ax = DatetimeIndex([], tz=tz)
 else:
 ax = ax.tz_localize(tz, ambiguous=ambiguous, nonexistent=nonexistent)
 return ax

if a level is given it must be a MultiIndex level or
equivalent to the axis name
if isinstance(ax, MultiIndex):
 level = ax._get_level_number(level)
 new_level = _tz_localize(ax.levels[level], tz, ambiguous, nonexistent)
 ax = ax.set_levels(new_level, level=level)
else:
 if level not in (None, 0, ax.name):
 raise ValueError(f"The level {level} is not valid")
 ax = _tz_localize(ax, tz, ambiguous, nonexistent)

result = self.copy(deep=copy)
result = result.set_axis(ax, axis=axis, inplace=False)
return result.__finalize__(self, method="tz_localize")

Numeric Methods
def abs(self: FrameOrSeries) -> FrameOrSeries:

```

```

"""
Return a Series/DataFrame with absolute numeric value of each element.

This function only applies to elements that are all numeric.

Returns

abs
 Series/DataFrame containing the absolute value of each element.

See Also

numpy.absolute : Calculate the absolute value element-wise.

Notes

For ``complex`` inputs, ``1.2 + 1j``, the absolute value is
:math:`\sqrt{a^2 + b^2}`.

Examples

Absolute numeric values in a Series.

>>> s = pd.Series([-1.10, 2, -3.33, 4])
>>> s.abs()
0 1.10
1 2.00
2 3.33
3 4.00
dtype: float64

Absolute numeric values in a Series with complex numbers.

>>> s = pd.Series([1.2 + 1j])
>>> s.abs()
0 1.56205
dtype: float64

Absolute numeric values in a Series with a Timedelta element.

>>> s = pd.Series([pd.Timedelta('1 days')])
>>> s.abs()
0 1 days
dtype: timedelta64[ns]

Select rows with data closest to certain value using argsort (from
`StackOverflow <https://stackoverflow.com/a/17758115>`__).

>>> df = pd.DataFrame({
... 'a': [4, 5, 6, 7],
... 'b': [10, 20, 30, 40],
... 'c': [100, 50, -30, -50]
... })
>>> df
 a b c
0 4 10 100
1 5 20 50
2 6 30 -30
3 7 40 -50
>>> df.loc[(df.c - 43).abs().argsort()]
 a b c
1 5 20 50
0 4 10 100

```

```

2 6 30 -30
3 7 40 -50
"""
return np.abs(self)

def describe(
 self: FrameOrSeries, percentiles=None, include=None, exclude=None
) -> FrameOrSeries:
"""
Generate descriptive statistics.

Descriptive statistics include those that summarize the central
tendency, dispersion and shape of a
dataset's distribution, excluding ``NaN`` values.

Analyzes both numeric and object series, as well
as ``DataFrame`` column sets of mixed data types. The output
will vary depending on what is provided. Refer to the notes
below for more detail.

Parameters

percentiles : list-like of numbers, optional
 The percentiles to include in the output. All should
 fall between 0 and 1. The default is
 ``[.25, .5, .75]``, which returns the 25th, 50th, and
 75th percentiles.
include : 'all', list-like of dtypes or None (default), optional
 A white list of data types to include in the result. Ignored
 for ``Series``. Here are the options:

 - 'all' : All columns of the input will be included in the output.
 - A list-like of dtypes : Limits the results to the
 provided data types.
 To limit the result to numeric types submit
 ``numpy.number``. To limit it instead to object columns submit
 the ``numpy.object`` data type. Strings
 can also be used in the style of
 ``select_dtypes`` (e.g. ``df.describe(include=['O'])``). To
 select pandas categorical columns, use ``'category'``
 - None (default) : The result will include all numeric columns.
exclude : list-like of dtypes or None (default), optional,
 A black list of data types to omit from the result. Ignored
 for ``Series``. Here are the options:

 - A list-like of dtypes : Excludes the provided data types
 from the result. To exclude numeric types submit
 ``numpy.number``. To exclude object columns submit the data
 type ``numpy.object``. Strings can also be used in the style of
 ``select_dtypes`` (e.g. ``df.describe(exclude=['O'])``). To
 exclude pandas categorical columns, use ``'category'``
 - None (default) : The result will exclude nothing.

Returns

Series or DataFrame
 Summary statistics of the Series or Dataframe provided.

See Also

DataFrame.count: Count number of non-NA/null observations.
DataFrame.max: Maximum of the values in the object.
DataFrame.min: Minimum of the values in the object.

```

```
DataFrame.mean: Mean of the values.
DataFrame.std: Standard deviation of the observations.
DataFrame.select_dtypes: Subset of a DataFrame including/excluding
 columns based on their dtype.
```

#### Notes

-----

For numeric data, the result's index will include ``count'', ``mean'', ``std'', ``min'', ``max'' as well as lower, ``50'' and upper percentiles. By default the lower percentile is ``25'' and the upper percentile is ``75''. The ``50'' percentile is the same as the median.

For object data (e.g. strings or timestamps), the result's index will include ``count'', ``unique'', ``top'', and ``freq''. The ``top'' is the most common value. The ``freq'' is the most common value's frequency. Timestamps also include the ``first'' and ``last'' items.

If multiple object values have the highest count, then the ``count'' and ``top'' results will be arbitrarily chosen from among those with the highest count.

For mixed data types provided via a ``DataFrame'', the default is to return only an analysis of numeric columns. If the dataframe consists only of object and categorical data without any numeric columns, the default is to return an analysis of both the object and categorical columns. If ``include='all'' is provided as an option, the result will include a union of attributes of each type.

The `include` and `exclude` parameters can be used to limit which columns in a ``DataFrame`` are analyzed for the output. The parameters are ignored when analyzing a ``Series``.

#### Examples

-----

Describing a numeric ``Series''.

```
>>> s = pd.Series([1, 2, 3])
>>> s.describe()
count 3.0
mean 2.0
std 1.0
min 1.0
25% 1.5
50% 2.0
75% 2.5
max 3.0
dtype: float64
```

Describing a categorical ``Series''.

```
>>> s = pd.Series(['a', 'a', 'b', 'c'])
>>> s.describe()
count 4
unique 3
top a
freq 2
dtype: object
```

Describing a timestamp ``Series''.

```
>>> s = pd.Series([
... np.datetime64("2000-01-01"),
```

```
... np.datetime64("2010-01-01"),
... np.datetime64("2010-01-01")
...])
>>> s.describe()
count 3
mean 2006-09-01 08:00:00
min 2000-01-01 00:00:00
25% 2004-12-31 12:00:00
50% 2010-01-01 00:00:00
75% 2010-01-01 00:00:00
max 2010-01-01 00:00:00
dtype: object
```

Describing a ``DataFrame``. By default only numeric fields are returned.

```
>>> df = pd.DataFrame({'categorical': pd.Categorical(['d', 'e', 'f']),
... 'numeric': [1, 2, 3],
... 'object': ['a', 'b', 'c']}
...)
>>> df.describe()
 numeric
count 3.0
mean 2.0
std 1.0
min 1.0
25% 1.5
50% 2.0
75% 2.5
max 3.0
```

Describing all columns of a ``DataFrame`` regardless of data type.

```
>>> df.describe(include='all') # doctest: +SKIP
 categorical numeric object
count 3 3.0 3
unique 3 NaN 3
top f NaN a
freq 1 NaN 1
mean NaN 2.0 NaN
std NaN 1.0 NaN
min NaN 1.0 NaN
25% NaN 1.5 NaN
50% NaN 2.0 NaN
75% NaN 2.5 NaN
max NaN 3.0 NaN
```

Describing a column from a ``DataFrame`` by accessing it as an attribute.

```
>>> df.numeric.describe()
count 3.0
mean 2.0
std 1.0
min 1.0
25% 1.5
50% 2.0
75% 2.5
max 3.0
Name: numeric, dtype: float64
```

Including only numeric columns in a ``DataFrame`` description.

```
>>> df.describe(include=[np.number])
 numeric
count 3.0
mean 2.0
std 1.0
min 1.0
25% 1.5
50% 2.0
75% 2.5
max 3.0

Including only string columns in a ``DataFrame`` description.

>>> df.describe(include=[np.object]) # doctest: +SKIP
 object
count 3
unique 3
top a
freq 1

Including only categorical columns from a ``DataFrame`` description.

>>> df.describe(include=['category'])
 categorical
count 3
unique 3
top f
freq 1

Excluding numeric columns from a ``DataFrame`` description.

>>> df.describe(exclude=[np.number]) # doctest: +SKIP
 categorical object
count 3 3
unique 3 3
top f a
freq 1 1

Excluding object columns from a ``DataFrame`` description.

>>> df.describe(exclude=[np.object]) # doctest: +SKIP
 categorical numeric
count 3 3.0
unique 3 NaN
top f NaN
freq 1 NaN
mean NaN 2.0
std NaN 1.0
min NaN 1.0
25% NaN 1.5
50% NaN 2.0
75% NaN 2.5
max NaN 3.0
"""

if self.ndim == 2 and self.columns.size == 0:
 raise ValueError("Cannot describe a DataFrame without columns")

if percentiles is not None:
 # explicit conversion of `percentiles` to list
 percentiles = list(percentiles)

 # get them all to be in [0, 1]
 validate_percentile(percentiles)
```

```

 # median should always be included
 if 0.5 not in percentiles:
 percentiles.append(0.5)
 percentiles = np.asarray(percentiles)
 else:
 percentiles = np.array([0.25, 0.5, 0.75])

 # sort and check for duplicates
 unique_pcts = np.unique(percentiles)
 if len(unique_pcts) < len(percentiles):
 raise ValueError("percentiles cannot contain duplicates")
 percentiles = unique_pcts

 formatted_percentiles = format_percentiles(percentiles)

def describe_numeric_1d(series):
 stat_index = (
 ["count", "mean", "std", "min"] + formatted_percentiles + ["max"]
)
 d = (
 [series.count(), series.mean(), series.std(), series.min()]
 + series.quantile(percentiles).tolist()
 + [series.max()]
)
 return pd.Series(d, index=stat_index, name=series.name)

def describe_categorical_1d(data):
 names = ["count", "unique"]
 objcounts = data.value_counts()
 count_unique = len(objcounts[objcounts != 0])
 result = [data.count(), count_unique]
 dtype = None
 if result[1] > 0:
 top, freq = objcounts.index[0], objcounts.iloc[0]
 names += ["top", "freq"]
 result += [top, freq]

 # If the DataFrame is empty, set 'top' and 'freq' to None
 # to maintain output shape consistency
 else:
 names += ["top", "freq"]
 result += [np.nan, np.nan]
 dtype = "object"

 return pd.Series(result, index=names, name=data.name, dtype=dtype)

def describe_timestamp_1d(data):
 # GH-30164
 stat_index = ["count", "mean", "min"] + formatted_percentiles + ["max"]
 d = (
 [data.count(), data.mean(), data.min()]
 + data.quantile(percentiles).tolist()
 + [data.max()]
)
 return pd.Series(d, index=stat_index, name=data.name)

def describe_1d(data):
 if is_bool_dtype(data):
 return describe_categorical_1d(data)
 elif is_numeric_dtype(data):
 return describe_numeric_1d(data)
 elif is_datetime64_any_dtype(data):

```

```

 return describe_timestamp_1d(data)
 elif is_timedelta64_dtype(data):
 return describe_numeric_1d(data)
 else:
 return describe_categorical_1d(data)

if self.ndim == 1:
 return describe_1d(self)
elif (include is None) and (exclude is None):
 # when some numerics are found, keep only numerics
 data = self.select_dtypes(include=[np.number])
 if len(data.columns) == 0:
 data = self
elif include == "all":
 if exclude is not None:
 msg = "exclude must be None when include is 'all'"
 raise ValueError(msg)
 data = self
else:
 data = self.select_dtypes(include=include, exclude=exclude)

ldesc = [describe_1d(s) for _, s in data.items()]
set a convenient order for rows
names: List[Label] = []
ldesc_indexes = sorted((x.index for x in ldesc), key=len)
for idxnames in ldesc_indexes:
 for name in idxnames:
 if name not in names:
 names.append(name)

d = pd.concat([x.reindex(names, copy=False) for x in ldesc], axis=1, sort=False)
d.columns = data.columns.copy()
return d

```

**\_shared\_docs**

- "pct\_change"

**】 = """**

Percentage change between the current and a prior element.

Computes the percentage change from the immediately previous row by default. This is useful in comparing the percentage of change in a time series of elements.

**Parameters**

-----

**periods** : int, default 1  
     Periods to shift for forming percent change.

**fill\_method** : str, default 'pad'  
     How to handle NAs before computing percent changes.

**limit** : int, default None  
     The number of consecutive NAs to fill before stopping.

**freq** : DateOffset, timedelta, or str, optional  
     Increment to use from time series API (e.g. 'M' or BDay()).

**\*\*kwargs**  
     Additional keyword arguments are passed into  
     `DataFrame.shift` or `Series.shift`.

## Returns

-----

**chg** : Series or DataFrame

The same type as the calling object.

## See Also

```

Series.diff : Compute the difference of two elements in a Series.
DataFrame.diff : Compute the difference of two elements in a DataFrame.
Series.shift : Shift the index by some number of periods.
DataFrame.shift : Shift the index by some number of periods.
```

#### Examples

```

Series
```

```
>>> s = pd.Series([90, 91, 85])
>>> s
0 90
1 91
2 85
dtype: int64
```

```
>>> s.pct_change()
0 NaN
1 0.011111
2 -0.065934
dtype: float64
```

```
>>> s.pct_change(periods=2)
0 NaN
1 NaN
2 -0.055556
dtype: float64
```

See the percentage change in a Series where filling NAs with last valid observation forward to next valid.

```
>>> s = pd.Series([90, 91, None, 85])
>>> s
0 90.0
1 91.0
2 NaN
3 85.0
dtype: float64

>>> s.pct_change(fill_method='ffill')
0 NaN
1 0.011111
2 0.000000
3 -0.065934
dtype: float64
```

```
DataFrame
```

Percentage change in French franc, Deutsche Mark, and Italian lira from 1980-01-01 to 1980-03-01.

```
>>> df = pd.DataFrame({
... 'FR': [4.0405, 4.0963, 4.3149],
... 'GR': [1.7246, 1.7482, 1.8519],
... 'IT': [804.74, 810.01, 860.13]},
... index=['1980-01-01', '1980-02-01', '1980-03-01'])
>>> df
 FR GR IT
1980-01-01 4.0405 1.7246 804.74
1980-02-01 4.0963 1.7482 810.01
1980-03-01 4.3149 1.8519 860.13
```

```

>>> df.pct_change()
 FR GR IT
1980-01-01 NaN NaN NaN
1980-02-01 0.013810 0.013684 0.006549
1980-03-01 0.053365 0.059318 0.061876

Percentage of change in GOOG and APPL stock volume. Shows computing
the percentage change between columns.

>>> df = pd.DataFrame({
... '2016': [1769950, 30586265],
... '2015': [1500923, 40912316],
... '2014': [1371819, 41403351],
... index=['GOOG', 'APPL'])
>>> df
 2016 2015 2014
GOOG 1769950 1500923 1371819
APPL 30586265 40912316 41403351

>>> df.pct_change(axis='columns')
 2016 2015 2014
GOOG NaN -0.151997 -0.086016
APPL NaN 0.337604 0.012002
"""

@Appender(_shared_docs["pct_change"] % _shared_doc_kwargs)
def pct_change(
 self: FrameOrSeries,
 periods=1,
 fill_method="pad",
 limit=None,
 freq=None,
 **kwargs,
) -> FrameOrSeries:
 # TODO: Not sure if above is correct - need someone to confirm.
 axis = self._get_axis_number(kwargs.pop("axis", self._stat_axis_name))
 if fill_method is None:
 data = self
 else:
 _data = self.fillna(method=fill_method, axis=axis, limit=limit)
 assert _data is not None # needed for mypy
 data = _data

 rs = data.div(data.shift(periods=periods, freq=freq, axis=axis, **kwargs)) - 1
 if freq is not None:
 # Shift method is implemented differently when freq is not None
 # We want to restore the original index
 rs = rs.loc[~rs.index.duplicated()]
 rs = rs.reindex_like(data)
 return rs

def _agg_by_level(self, name, axis=0, level=0, skipna=True, **kwargs):
 if axis is None:
 raise ValueError("Must specify 'axis' when aggregating by level.")
 grouped = self.groupby(level=level, axis=axis, sort=False)
 if hasattr(grouped, name) and skipna:
 return getattr(grouped, name)(**kwargs)
 axis = self._get_axis_number(axis)
 method = getattr(type(self), name)
 applyf = lambda x: method(x, axis=axis, skipna=skipna, **kwargs)
 return grouped.aggregate(applyf)

@classmethod

```

```

def _add_numeric_operations(cls):
 """
 Add the operations to the cls; evaluate the doc strings again
 """
 axis_descr, name1, name2 = _doc_parms(cls)

 cls.any = _make_logical_function(
 cls,
 "any",
 name1=name1,
 name2=name2,
 axis_descr=axis_descr,
 desc=_any_desc,
 func=nanops.nanany,
 see_also=_any_see_also,
 examples=_any_examples,
 empty_value=False,
)
 cls.all = _make_logical_function(
 cls,
 "all",
 name1=name1,
 name2=name2,
 axis_descr=axis_descr,
 desc=_all_desc,
 func=nanops.nanall,
 see_also=_all_see_also,
 examples=_all_examples,
 empty_value=True,
)
)

 @Substitution(
 desc="Return the mean absolute deviation of the values "
 "for the requested axis.",
 name1=name1,
 name2=name2,
 axis_descr=axis_descr,
 min_count="",
 see_also="",
 examples="",
)
 @_Appender(_num_doc_mad)
def mad(self, axis=None, skipna=None, level=None):
 if skipna is None:
 skipna = True
 if axis is None:
 axis = self._stat_axis_number
 if level is not None:
 return self._agg_by_level("mad", axis=axis, level=level, skipna=skipna)

 data = self._get_numeric_data()
 if axis == 0:
 demeaned = data - data.mean(axis=0)
 else:
 demeaned = data.sub(data.mean(axis=1), axis=0)
 return np.abs(demeaned).mean(axis=axis, skipna=skipna)

cls.mad = mad

cls.sem = _make_stat_function_ddof(
 cls,
 "sem",
 name1=name1,

```

```
 name2=name2,
 axis_descr=axis_descr,
 desc="Return unbiased standard error of the mean over requested "
 "axis.\n\nNormalized by N-1 by default. This can be changed "
 "using the ddof argument",
 func=nanops.nansem,
)
 cls.var = _make_stat_function_ddof(
 cls,
 "var",
 name1=name1,
 name2=name2,
 axis_descr=axis_descr,
 desc="Return unbiased variance over requested axis.\n\nNormalized by "
 "N-1 by default. This can be changed using the ddof argument",
 func=nanops.nanvar,
)
 cls.std = _make_stat_function_ddof(
 cls,
 "std",
 name1=name1,
 name2=name2,
 axis_descr=axis_descr,
 desc="Return sample standard deviation over requested axis."
 "\n\nNormalized by N-1 by default. This can be changed using the "
 "ddof argument",
 func=nanops.nanstd,
)

 cls.cummin = _make_cum_function(
 cls,
 "cummin",
 name1=name1,
 name2=name2,
 axis_descr=axis_descr,
 desc="minimum",
 accum_func=np.minimum.accumulate,
 accum_func_name="min",
 examples=_cummin_examples,
)
 cls.cumsum = _make_cum_function(
 cls,
 "cumsum",
 name1=name1,
 name2=name2,
 axis_descr=axis_descr,
 desc="sum",
 accum_func=np.cumsum,
 accum_func_name="sum",
 examples=_cumsum_examples,
)
 cls.cumprod = _make_cum_function(
 cls,
 "cumprod",
 name1=name1,
 name2=name2,
 axis_descr=axis_descr,
 desc="product",
 accum_func=np.cumprod,
 accum_func_name="prod",
 examples=_cumprod_examples,
)
 cls.cummax = _make_cum_function(
```

```
 cls,
 "cummax",
 name1=name1,
 name2=name2,
 axis_descr=axis_descr,
 desc="maximum",
 accum_func=np.maximum.accumulate,
 accum_func_name="max",
 examples=_cummax_examples,
)

cls.sum = _make_min_count_stat_function(
 cls,
 "sum",
 name1=name1,
 name2=name2,
 axis_descr=axis_descr,
 desc="Return the sum of the values for the requested axis.\n\n"
 "This is equivalent to the method ``numpy.sum``.",
 func=nanops.nansum,
 see_also=_stat_func_see_also,
 examples=_sum_examples,
)
cls.mean = _make_stat_function(
 cls,
 "mean",
 name1=name1,
 name2=name2,
 axis_descr=axis_descr,
 desc="Return the mean of the values for the requested axis.",
 func=nanops.nanmean,
)
cls.skew = _make_stat_function(
 cls,
 "skew",
 name1=name1,
 name2=name2,
 axis_descr=axis_descr,
 desc="Return unbiased skew over requested axis.\n\nNormalized by N-1.",
 func=nanops.nanskew,
)
cls.kurt = _make_stat_function(
 cls,
 "kurt",
 name1=name1,
 name2=name2,
 axis_descr=axis_descr,
 desc="Return unbiased kurtosis over requested axis.\n\n"
 "Kurtosis obtained using Fisher's definition of\n"
 "kurtosis (kurtosis of normal == 0.0). Normalized "
 "by N-1.",
 func=nanops.nankurt,
)
cls.kurtosis = cls.kurt
cls.prod = _make_min_count_stat_function(
 cls,
 "prod",
 name1=name1,
 name2=name2,
 axis_descr=axis_descr,
 desc="Return the product of the values for the requested axis.",
 func=nanops.nanprod,
 examples=_prod_examples,
```

```
)
cls.product = cls.prod
cls.median = _make_stat_function(
 cls,
 "median",
 name1=name1,
 name2=name2,
 axis_descr=axis_descr,
 desc="Return the median of the values for the requested axis.",
 func=nanops.nanmedian,
)
cls.max = _make_stat_function(
 cls,
 "max",
 name1=name1,
 name2=name2,
 axis_descr=axis_descr,
 desc="Return the maximum of the values for the requested axis.\n\n"
 "If you want the *index* of the maximum, use ``idxmax``. This is"
 "the equivalent of the ``numpy.ndarray`` method ``argmax``.",
 func=nanops.nanmax,
 see_also=_stat_func_see_also,
 examples=_max_examples,
)
cls.min = _make_stat_function(
 cls,
 "min",
 name1=name1,
 name2=name2,
 axis_descr=axis_descr,
 desc="Return the minimum of the values for the requested axis.\n\n"
 "If you want the *index* of the minimum, use ``idxmin``. This is"
 "the equivalent of the ``numpy.ndarray`` method ``argmin``.",
 func=nanops.nanmin,
 see_also=_stat_func_see_also,
 examples=_min_examples,
)

@classmethod
def _add_series_or_dataframe_operations(cls):
 """
 Add the series or dataframe only operations to the cls; evaluate
 the doc strings again.
 """
 from pandas.core.window import EWM, Expanding, Rolling, Window

 @doc(Rolling)
 def rolling(
 self,
 window,
 min_periods=None,
 center=False,
 win_type=None,
 on=None,
 axis=0,
 closed=None,
):
 axis = self._get_axis_number(axis)

 if win_type is not None:
 return Window(
 self,
 window=window,
)
```

```

 min_periods=min_periods,
 center=center,
 win_type=win_type,
 on=on,
 axis=axis,
 closed=closed,
)

 return Rolling(
 self,
 window=window,
 min_periods=min_periods,
 center=center,
 win_type=win_type,
 on=on,
 axis=axis,
 closed=closed,
)

cls.rolling = rolling

@doc(Expanding)
def expanding(self, min_periods=1, center=False, axis=0):
 axis = self._get_axis_number(axis)
 return Expanding(self, min_periods=min_periods, center=center, axis=axis)

cls.expanding = expanding

@doc(EWM)
def ewm(
 self,
 com=None,
 span=None,
 halflife=None,
 alpha=None,
 min_periods=0,
 adjust=True,
 ignore_na=False,
 axis=0,
):
 axis = self._get_axis_number(axis)
 return EWM(
 self,
 com=com,
 span=span,
 halflife=halflife,
 alpha=alpha,
 min_periods=min_periods,
 adjust=adjust,
 ignore_na=ignore_na,
 axis=axis,
)

cls.ewm = ewm

@Appender(_shared_docs["transform"] % dict(axis="", **_shared_doc_kwargs))
def transform(self, func, *args, **kwargs):
 result = self.agg(func, *args, **kwargs)
 if is_scalar(result) or len(result) != len(self):
 raise ValueError("transforms cannot produce aggregated results")

 return result

```

```

Misc methods

_shared_docs[
 "valid_index"
] = """
 Return index for %(position)s non-NA/null value.

 Returns

 scalar : type of index

 Notes

 If all elements are non-NA/null, returns None.
 Also returns None for empty %(klass)s.
"""

def _find_valid_index(self, how: str):
 """
 Retrieves the index of the first valid value.

 Parameters

 how : {'first', 'last'}
 Use this parameter to change between the first or last valid index.

 Returns

 idx_first_valid : type of index
 """
 idxpos = find_valid_index(self._values, how)
 if idxpos is None:
 return None
 return self.index[idxpos]

@Appender(
 _shared_docs["valid_index"] % {"position": "first", "klass": "Series/DataFrame"}
)
def first_valid_index(self):
 return self._find_valid_index("first")

@Appender(
 _shared_docs["valid_index"] % {"position": "last", "klass": "Series/DataFrame"}
)
def last_valid_index(self):
 return self._find_valid_index("last")

def _doc_parms(cls):
 """Return a tuple of the doc parms."""
 axis_descr = (
 f"{{{', '.join(f'{a} ({i})' for i, a in enumerate(cls._AXIS_ORDERS))}}}"
)
 name = cls._constructor_sliced.__name__ if cls._AXIS_LEN > 1 else "scalar"
 name2 = cls.__name__
 return axis_descr, name, name2

 _num_doc = """
%(desc)s

Parameters

```

```

axis : %(axis_descr)s
 Axis for the function to be applied on.
skipna : bool, default True
 Exclude NA/null values when computing the result.
level : int or level name, default None
 If the axis is a MultiIndex (hierarchical), count along a
 particular level, collapsing into a %(name1)s.
numeric_only : bool, default None
 Include only float, int, boolean columns. If None, will attempt to use
 everything, then use only numeric data. Not implemented for Series.
%(min_count)s\
**kwargs
 Additional keyword arguments to be passed to the function.
```

Returns

```

%(name1)s or %(name2)s (if level specified) \
%(see_also)s\
%(examples)s
"""
```

```
_num_doc_mad = """
%(desc)s
```

Parameters

```

axis : %(axis_descr)s
 Axis for the function to be applied on.
skipna : bool, default None
 Exclude NA/null values when computing the result.
level : int or level name, default None
 If the axis is a MultiIndex (hierarchical), count along a
 particular level, collapsing into a %(name1)s.
```

Returns

```

%(name1)s or %(name2)s (if level specified) \
%(see_also)s\
%(examples)s
"""
```

```
_num_ddof_doc = """
%(desc)s
```

Parameters

```

axis : %(axis_descr)s
skipna : bool, default True
 Exclude NA/null values. If an entire row/column is NA, the result
 will be NA.
level : int or level name, default None
 If the axis is a MultiIndex (hierarchical), count along a
 particular level, collapsing into a %(name1)s.
ddof : int, default 1
 Delta Degrees of Freedom. The divisor used in calculations is N - ddof,
 where N represents the number of elements.
numeric_only : bool, default None
 Include only float, int, boolean columns. If None, will attempt to use
 everything, then use only numeric data. Not implemented for Series.
```

Returns

```
%(%name1)s or %(%name2)s (if level specified)\n"""\n\n_bool_doc = """\n%(desc)s\n\nParameters\n-----\naxis : {0 or 'index', 1 or 'columns', None}, default 0\n Indicate which axis or axes should be reduced.\n\n * 0 / 'index' : reduce the index, return a Series whose index is the\n original column labels.\n * 1 / 'columns' : reduce the columns, return a Series whose index is the\n original index.\n * None : reduce all axes, return a scalar.\n\nbool_only : bool, default None\n Include only boolean columns. If None, will attempt to use everything,\n then use only boolean data. Not implemented for Series.\nskipna : bool, default True\n Exclude NA/null values. If the entire row/column is NA and skipna is\n True, then the result will be %(empty_value)s, as for an empty row/column.\n If skipna is False, then NA are treated as True, because these are not\n equal to zero.\nlevel : int or level name, default None\n If the axis is a MultiIndex (hierarchical), count along a\n particular level, collapsing into a %(name1)s.\n**kwargs : any, default None\n Additional keywords have no effect but might be accepted for\n compatibility with NumPy.\n\nReturns\n-----\n%(name1)s or %(%name2)s\n If level is specified, then, %(%name2)s is returned; otherwise, %(name1)s\n is returned.\n\n%(see_also)s\n%(examples)s"""\n\n_all_desc = """\\\nReturn whether all elements are True, potentially over an axis.\n\nReturns True unless there at least one element within a series or\nalong a Dataframe axis that is False or equivalent (e.g. zero or\nempty)."""\n\n_all_examples = """\\\nExamples\n-----\n**Series**\n\n>>> pd.Series([True, True]).all()\nTrue\n>>> pd.Series([True, False]).all()\nFalse\n>>> pd.Series([]).all()\nTrue\n>>> pd.Series([np.nan]).all()\nTrue\n>>> pd.Series([np.nan]).all(skipna=False)\nTrue
```

```
DataFrames

Create a dataframe from a dictionary.

>>> df = pd.DataFrame({'col1': [True, True], 'col2': [True, False]})
>>> df
 col1 col2
0 True True
1 True False

Default behaviour checks if column-wise values all return True.

>>> df.all()
 col1 True
 col2 False
 dtype: bool

Specify ``axis='columns'`` to check if row-wise values all return True.

>>> df.all(axis='columns')
0 True
1 False
dtype: bool

Or ``axis=None`` for whether every value is True.

>>> df.all(axis=None)
False
"""

_all_see_also = """\nSee Also\n-----\nSeries.all : Return True if all elements are True.\nDataFrame.any : Return True if one (or more) elements are True.\n"""

_cnum_doc = """\nReturn cumulative %(desc)s over a DataFrame or Series axis.

Returns a DataFrame or Series of the same size containing the cumulative
%(desc)s.

Parameters

axis : {0 or 'index', 1 or 'columns'}, default 0
 The index or the name of the axis. 0 is equivalent to None or 'index'.
skipna : bool, default True
 Exclude NA/null values. If an entire row/column is NA, the result
 will be NA.
*args, **kwargs
 Additional keywords have no effect but might be accepted for
 compatibility with NumPy.

Returns

%(name1)s or %(name2)s
 Return cumulative %(desc)s of %(name1)s or %(name2)s.

See Also

core.window.Expanding.%(accum_func_name)s : Similar functionality
 but ignores ``NaN`` values.
```

```

%(name2)s.%(_accum_func_name)s : Return the %(desc)s over
 %(name2)s axis.
%(name2)s.cummax : Return cumulative maximum over %(name2)s axis.
%(name2)s.cummin : Return cumulative minimum over %(name2)s axis.
%(name2)s.cumsum : Return cumulative sum over %(name2)s axis.
%(name2)s.cumprod : Return cumulative product over %(name2)s axis.

%(examples)s"""

_cummin_examples = """\
Examples

Series

>>> s = pd.Series([2, np.nan, 5, -1, 0])
>>> s
0 2.0
1 NaN
2 5.0
3 -1.0
4 0.0
dtype: float64

By default, NA values are ignored.

>>> s.cummin()
0 2.0
1 NaN
2 2.0
3 -1.0
4 -1.0
dtype: float64

To include NA values in the operation, use ``skipna=False``

>>> s.cummin(skipna=False)
0 2.0
1 NaN
2 NaN
3 NaN
4 NaN
dtype: float64

DataFrame

>>> df = pd.DataFrame([[2.0, 1.0],
... [3.0, np.nan],
... [1.0, 0.0]],
... columns=list('AB'))
>>> df
 A B
0 2.0 1.0
1 3.0 NaN
2 1.0 0.0

By default, iterates over rows and finds the minimum
in each column. This is equivalent to ``axis=None`` or ``axis='index'``.

>>> df.cummin()
 A B
0 2.0 1.0
1 2.0 NaN
2 1.0 0.0

```

```
To iterate over columns and find the minimum in each row,
use ``axis=1``
```

```
>>> df.cummin(axis=1)
 A B
0 2.0 1.0
1 3.0 NaN
2 1.0 0.0
"""

_cumsum_examples = """\brExamples

Series

>>> s = pd.Series([2, np.nan, 5, -1, 0])
>>> s
0 2.0
1 NaN
2 5.0
3 -1.0
4 0.0
dtype: float64
```

By default, NA values are ignored.

```
>>> s.cumsum()
0 2.0
1 NaN
2 7.0
3 6.0
4 6.0
dtype: float64
```

To include NA values in the operation, use ``skipna=False``

```
>>> s.cumsum(skipna=False)
0 2.0
1 NaN
2 NaN
3 NaN
4 NaN
dtype: float64
```

\*\*DataFrame\*\*

```
>>> df = pd.DataFrame([[2.0, 1.0],
... [3.0, np.nan],
... [1.0, 0.0]],
... columns=list('AB'))
>>> df
 A B
0 2.0 1.0
1 3.0 NaN
2 1.0 0.0
```

By default, iterates over rows and finds the sum  
in each column. This is equivalent to ``axis=None`` or ``axis='index'``.

```
>>> df.cumsum()
 A B
0 2.0 1.0
```

```

1 5.0 NaN
2 6.0 1.0

To iterate over columns and find the sum in each row,
use ``axis=1``

>>> df.cumsum(axis=1)
 A B
0 2.0 3.0
1 3.0 NaN
2 1.0 1.0
"""

_cumprod_examples = """\
Examples

Series

>>> s = pd.Series([2, np.nan, 5, -1, 0])
>>> s
0 2.0
1 NaN
2 5.0
3 -1.0
4 0.0
dtype: float64

By default, NA values are ignored.

>>> s.cumprod()
0 2.0
1 NaN
2 10.0
3 -10.0
4 -0.0
dtype: float64

To include NA values in the operation, use ``skipna=False``

>>> s.cumprod(skipna=False)
0 2.0
1 NaN
2 NaN
3 NaN
4 NaN
dtype: float64

DataFrame

>>> df = pd.DataFrame([[2.0, 1.0],
... [3.0, np.nan],
... [1.0, 0.0]],
... columns=list('AB'))
>>> df
 A B
0 2.0 1.0
1 3.0 NaN
2 1.0 0.0

By default, iterates over rows and finds the product
in each column. This is equivalent to ``axis=None`` or ``axis='index'``.

>>> df.cumprod()

```

```
A B
0 2.0 1.0
1 6.0 NaN
2 6.0 0.0
```

To iterate over columns and find the product in each row,  
use ``axis=1``

```
>>> df.cumprod(axis=1)
A B
0 2.0 2.0
1 3.0 NaN
2 1.0 0.0
"""

_cummax_examples = """\
Examples

Series

>>> s = pd.Series([2, np.nan, 5, -1, 0])
>>> s
0 2.0
1 NaN
2 5.0
3 -1.0
4 0.0
dtype: float64
```

By default, NA values are ignored.

```
>>> s.cummax()
0 2.0
1 NaN
2 5.0
3 5.0
4 5.0
dtype: float64
```

To include NA values in the operation, use ``skipna=False``

```
>>> s.cummax(skipna=False)
0 2.0
1 NaN
2 NaN
3 NaN
4 NaN
dtype: float64
```

\*\*DataFrame\*\*

```
>>> df = pd.DataFrame([[2.0, 1.0],
... [3.0, np.nan],
... [1.0, 0.0]],
... columns=list('AB'))
>>> df
A B
0 2.0 1.0
1 3.0 NaN
2 1.0 0.0
```

By default, iterates over rows and finds the maximum  
in each column. This is equivalent to ``axis=None`` or ``axis='index'``.

```

>>> df.cummax()
 A B
0 2.0 1.0
1 3.0 NaN
2 3.0 1.0

To iterate over columns and find the maximum in each row,
use ``axis=1``

>>> df.cummax(axis=1)
 A B
0 2.0 2.0
1 3.0 NaN
2 1.0 1.0
"""

_any_see_also = """\
See Also

numpy.any : Numpy version of this method.
Series.any : Return whether any element is True.
Series.all : Return whether all elements are True.
DataFrame.any : Return whether any element is True over requested axis.
DataFrame.all : Return whether all elements are True over requested axis.
"""

_any_desc = """\
Return whether any element is True, potentially over an axis.

Returns False unless there at least one element within a series or
along a Dataframe axis that is True or equivalent (e.g. non-zero or
non-empty)."""

_any_examples = """\
Examples

Series

For Series input, the output is a scalar indicating whether any element
is True.

>>> pd.Series([False, False]).any()
False
>>> pd.Series([True, False]).any()
True
>>> pd.Series([]).any()
False
>>> pd.Series([np.nan]).any()
False
>>> pd.Series([np.nan]).any(skipna=False)
True

DataFrame

Whether each column contains at least one True element (the default).

>>> df = pd.DataFrame({"A": [1, 2], "B": [0, 2], "C": [0, 0]})
>>> df
 A B C
0 1 0 0
1 2 2 0

```

```
>>> df.any()
A True
B True
C False
dtype: bool

Aggregating over the columns.

>>> df = pd.DataFrame({"A": [True, False], "B": [1, 2]})
>>> df
 A B
0 True 1
1 False 2

>>> df.any(axis='columns')
0 True
1 True
dtype: bool

>>> df = pd.DataFrame({"A": [True, False], "B": [1, 0]})
>>> df
 A B
0 True 1
1 False 0

>>> df.any(axis='columns')
0 True
1 False
dtype: bool

Aggregating over the entire DataFrame with ``axis=None``.

>>> df.any(axis=None)
True

`any` for an empty DataFrame is an empty Series.

>>> pd.DataFrame([]).any()
Series([], dtype: bool)
"""

_shared_docs[
 "stat_func_example"
] = """

Examples

>>> idx = pd.MultiIndex.from_arrays([
... ['warm', 'warm', 'cold', 'cold'],
... ['dog', 'falcon', 'fish', 'spider']],
... names=['blooded', 'animal'])
>>> s = pd.Series([4, 2, 0, 8], name='legs', index=idx)
>>> s
blooded animal
warm dog 4
 falcon 2
cold fish 0
 spider 8
Name: legs, dtype: int64

>>> s.{stat_func}()
{default_output}
```

```
{verb} using level names, as well as indices.

>>> s.{stat_func}(level='blooded')
blooded
warm {level_output_0}
cold {level_output_1}
Name: legs, dtype: int64

>>> s.{stat_func}(level=0)
blooded
warm {level_output_0}
cold {level_output_1}
Name: legs, dtype: int64"""

_sum_examples = _shared_docs["stat_func_example"].format(
 stat_func="sum", verb="Sum", default_output=14, level_output_0=6, level_output_1=8
)

_sum_examples += """

By default, the sum of an empty or all-NA Series is ``0``.

>>> pd.Series([]).sum() # min_count=0 is the default
0.0

This can be controlled with the ``min_count`` parameter. For example, if
you'd like the sum of an empty series to be NaN, pass ``min_count=1``.

>>> pd.Series([]).sum(min_count=1)
nan

Thanks to the ``skipna`` parameter, ``min_count`` handles all-NA and
empty series identically.

>>> pd.Series([np.nan]).sum()
0.0

>>> pd.Series([np.nan]).sum(min_count=1)
nan"""

_max_examples = _shared_docs["stat_func_example"].format(
 stat_func="max", verb="Max", default_output=8, level_output_0=4, level_output_1=8
)

_min_examples = _shared_docs["stat_func_example"].format(
 stat_func="min", verb="Min", default_output=0, level_output_0=2, level_output_1=0
)

_stat_func_see_also = """

See Also

Series.sum : Return the sum.
Series.min : Return the minimum.
Series.max : Return the maximum.
Series.idxmin : Return the index of the minimum.
Series.idxmax : Return the index of the maximum.
DataFrame.sum : Return the sum over the requested axis.
DataFrame.min : Return the minimum over the requested axis.
DataFrame.max : Return the maximum over the requested axis.
DataFrame.idxmin : Return the index of the minimum over the requested axis.
DataFrame.idxmax : Return the index of the maximum over the requested axis."""

```

```
_prod_examples = """\n\nExamples\n-----\nBy default, the product of an empty or all-NA Series is ``1``\n\n>>> pd.Series([]).prod()\n1.0\n\nThis can be controlled with the ``min_count`` parameter\n\n>>> pd.Series([]).prod(min_count=1)\nnan\n\nThanks to the ``skipna`` parameter, ``min_count`` handles all-NA and\nempty series identically.\n\n>>> pd.Series([np.nan]).prod()\n1.0\n\n>>> pd.Series([np.nan]).prod(min_count=1)\nnan"""\n\n_min_count_stub = """\\nmin_count : int, default 0\n The required number of valid values to perform the operation. If fewer than\n ``min_count`` non-NA values are present the result will be NA.\n\n.. versionadded:: 0.22.0\n\n Added with the default being 0. This means the sum of an all-NA\n or empty Series is 0, and the product of an all-NA or empty\n Series is 1.\n"""\n\n\ndef _make_min_count_stat_function(\n cls,\n name: str,\n name1: str,\n name2: str,\n axis_descr: str,\n desc: str,\n func: Callable,\n see_also: str = "",\n examples: str = "",\n) -> Callable:\n @Substitution(\n desc=desc,\n name1=name1,\n name2=name2,\n axis_descr=axis_descr,\n min_count=_min_count_stub,\n see_also=see_also,\n examples=examples,\n)\n @Appender(_num_doc)\n def stat_func(\n self,\n axis=None,\n skipna=None,\n level=None,\n numeric_only=None,\n):\n pass\n\nstat = _make_min_count_stat_function(pd.Series)\n\n# These are here to prevent flake8 from complaining about\n# unused imports.\nsum = stat(func=Series.sum)\nprod = stat(func=Series.prod)
```

```

 min_count=0,
 **kwargs,
) :
 if name == "sum":
 nv.validate_sum(tuple(), kwargs)
 elif name == "prod":
 nv.validate_prod(tuple(), kwargs)
 else:
 nv.validate_stat_func(tuple(), kwargs, fname=name)
 if skipna is None:
 skipna = True
 if axis is None:
 axis = self._stat_axis_number
 if level is not None:
 return self._agg_by_level(
 name, axis=axis, level=level, skipna=skipna, min_count=min_count
)
 return self._reduce(
 func,
 name=name,
 axis=axis,
 skipna=skipna,
 numeric_only=numeric_only,
 min_count=min_count,
)

 return set_function_name(stat_func, name, cls)

def _make_stat_function(
 cls,
 name: str,
 name1: str,
 name2: str,
 axis_descr: str,
 desc: str,
 func: Callable,
 see_also: str = "",
 examples: str = "",
) -> Callable:
 @Substitution(
 desc=desc,
 name1=name1,
 name2=name2,
 axis_descr=axis_descr,
 min_count="",
 see_also=see_also,
 examples=examples,
)
 @Appender(_num_doc)
 def stat_func(
 self, axis=None, skipna=None, level=None, numeric_only=None, **kwargs
):
 if name == "median":
 nv.validate_median(tuple(), kwargs)
 else:
 nv.validate_stat_func(tuple(), kwargs, fname=name)
 if skipna is None:
 skipna = True
 if axis is None:
 axis = self._stat_axis_number
 if level is not None:
 return self._agg_by_level(name, axis=axis, level=level, skipna=skipna)

```

```

 return self._reduce(
 func, name=name, axis=axis, skipna=skipna, numeric_only=numeric_only
)

 return set_function_name(stat_func, name, cls)

def _make_stat_function_ddof(
 cls, name: str, name1: str, name2: str, axis_descr: str, desc: str, func: Callable
) -> Callable:
 @Substitution(desc=desc, name1=name1, name2=name2, axis_descr=axis_descr)
 @Appender(_num_ddof_doc)
 def stat_func(
 self, axis=None, skipna=None, level=None, ddof=1, numeric_only=None, **kwargs
):
 nv.validate_stat_ddof_func(tuple(), kwargs, fname=name)
 if skipna is None:
 skipna = True
 if axis is None:
 axis = self._stat_axis_number
 if level is not None:
 return self._agg_by_level(
 name, axis=axis, level=level, skipna=skipna, ddof=ddof
)
 return self._reduce(
 func, name, axis=axis, numeric_only=numeric_only, skipna=skipna, ddof=ddof
)

 return set_function_name(stat_func, name, cls)

def _make_cum_function(
 cls,
 name: str,
 name1: str,
 name2: str,
 axis_descr: str,
 desc: str,
 accum_func: Callable,
 accum_func_name: str,
 examples: str,
) -> Callable:
 @Substitution(
 desc=desc,
 name1=name1,
 name2=name2,
 axis_descr=axis_descr,
 accum_func_name=accum_func_name,
 examples=examples,
)
 @Appender(_cnum_doc)
 def cum_func(self, axis=None, skipna=True, *args, **kwargs):
 skipna = nv.validate_cum_func_with_skipna(skipna, args, kwargs, name)
 if axis is None:
 axis = self._stat_axis_number
 else:
 axis = self._get_axis_number(axis)

 if axis == 1:
 return cum_func(self.T, axis=0, skipna=skipna, *args, **kwargs).T

 def block_accum_func(blk_values):
 values = blk_values.T if hasattr(blk_values, "T") else blk_values

```

```

 result = nanops.na_accum_func(values, accum_func, skipna=skipna)

 result = result.T if hasattr(result, "T") else result
 return result

 result = self._mgr.apply(block_accum_func)

 return self._constructor(result).__finalize__(self, method=name)

return set_function_name(cum_func, name, cls)

def _make_logical_function(
 cls,
 name: str,
 name1: str,
 name2: str,
 axis_descr: str,
 desc: str,
 func: Callable,
 see_also: str,
 examples: str,
 empty_value: bool,
) -> Callable:
 @Substitution(
 desc=desc,
 name1=name1,
 name2=name2,
 axis_descr=axis_descr,
 see_also=see_also,
 examples=examples,
 empty_value=empty_value,
)
 @Appender(_bool_doc)
 def logical_func(self, axis=0, bool_only=None, skipna=True, level=None, **kwargs):
 nv.validate_logical_func(tuple(), kwargs, fname=name)
 if level is not None:
 if bool_only is not None:
 raise NotImplementedError(
 "Option bool_only is not implemented with option level."
)
 return self._agg_by_level(name, axis=axis, level=level, skipna=skipna)
 return self._reduce(
 func,
 name=name,
 axis=axis,
 skipna=skipna,
 numeric_only=bool_only,
 filter_type="bool",
)

return set_function_name(logical_func, name, cls)

```

<https://github.com/matplotlib/matplotlib/blob/master/lib/matplotlib/pyplot.py>

```
Note: The first part of this file can be modified in place, but the latter
part is autogenerated by the boilerplate.py script.

"""
`matplotlib.pyplot` is a state-based interface to matplotlib. It provides
a MATLAB-like way of plotting.

pyplot is mainly intended for interactive plots and simple cases of
programmatic plot generation:::

 import numpy as np
 import matplotlib.pyplot as plt

 x = np.arange(0, 5, 0.1)
 y = np.sin(x)
 plt.plot(x, y)

The object-oriented API is recommended for more complex plots.
"""

import functools
import importlib
import inspect
import logging
from numbers import Number
import re
import sys
import time
try:
 import threading
except ImportError:
 import dummy_threading as threading

from cycler import cycler
import matplotlib
import matplotlib.colorbar
import matplotlib.image
from matplotlib import rcsetup, style
from matplotlib import _pylab_helpers, interactive
from matplotlib import cbook
from matplotlib import docstring
from matplotlib.backend_bases import FigureCanvasBase, MouseButton
from matplotlib.figure import Figure, figaspect
from matplotlib.gridspec import GridSpec
from matplotlib import rcParams, rcParamsDefault, get_backend, rcParamsOrig
from matplotlib.rcsetup import interactive_bk as _interactive_bk
from matplotlib.artist import Artist
from matplotlib.axes import Axes, Subplot
from matplotlib.projections import PolarAxes
from matplotlib import mlab # for detrend_none, window_hanning
from matplotlib.scale import get_scale_docs, get_scale_names

from matplotlib import cm
from matplotlib.cm import get_cmap, register_cmap
```

```

import numpy as np

We may not need the following imports here:
from matplotlib.colors import Normalize
from matplotlib.lines import Line2D
from matplotlib.text import Text, Annotation
from matplotlib.patches import Polygon, Rectangle, Circle, Arrow
from matplotlib.widgets import SubplotTool, Button, Slider, Widget

from .ticker import (
 TickHelper, Formatter, FixedFormatter, NullFormatter, FuncFormatter,
 FormatStrFormatter, ScalarFormatter, LogFormatter, LogFormatterExponent,
 LogFormatterMathtext, Locator, IndexLocator, FixedLocator, NullLocator,
 LinearLocator, LogLocator, AutoLocator, MultipleLocator, MaxNLocator)

_log = logging.getLogger(__name__)

_code_objs = {
 cbook._rename_parameter:
 cbook._rename_parameter("", "old", "new", lambda new: None).__code__,
 cbook._make_keyword_only:
 cbook._make_keyword_only("", "p", lambda p: None).__code__,
}
}

def _copy_docstring_and_deprecators(method, func=None):
 if func is None:
 return functools.partial(_copy_docstring_and_deprecators, method)
 decorators = [docstring.copy(method)]
 # Check whether the definition of *method* includes _rename_parameter or
 # _make_keyword_only decorators; if so, propagate them to the pyplot
 # wrapper as well.
 while getattr(method, "__wrapped__", None) is not None:
 for decorator_maker, code in _code_objs.items():
 if method.__code__ is code:
 kwargs = {
 k: v.cell_contents
 for k, v in zip(code.co_freevars, method.__closure__)
 }
 assert kwargs["func"] is method.__wrapped__
 kwargs.pop("func")
 decorators.append(decorator_maker(**kwargs))
 method = method.__wrapped__
 for decorator in decorators[::-1]:
 func = decorator(func)
 return func

Global

_IP_REGISTERED = None
_INSTALL FIG_OBSERVER = False

def install_repl_displayhook():
 """
 Install a repl display hook so that any stale figure are automatically
 redrawn when control is returned to the repl.

 This works both with IPython and with vanilla python shells.
 """
 global _IP_REGISTERED

```

```

global _INSTALL FIG_OBSERVER

class _NotIPython(Exception):
 pass

see if we have IPython hooks around, if use them

try:
 if 'IPython' in sys.modules:
 from IPython import get_ipython
 ip = get_ipython()
 if ip is None:
 raise _NotIPython()

 if _IP_REGISTERED:
 return

 def post_execute():
 if matplotlib.is_interactive():
 draw_all()

 # IPython >= 2
 try:
 ip.events.register('post_execute', post_execute)
 except AttributeError:
 # IPython 1.x
 ip.register_post_execute(post_execute)

 _IP_REGISTERED = post_execute
 _INSTALL FIG_OBSERVER = False

 # trigger IPython's eventloop integration, if available
 from IPython.core.pylabtools import backend2gui

 ipython_gui_name = backend2gui.get(get_backend())
 if ipython_gui_name:
 ip.enable_gui(ipython_gui_name)
 else:
 _INSTALL FIG_OBSERVER = True

import failed or ipython is not running
except (ImportError, _NotIPython):
 _INSTALL FIG_OBSERVER = True

def uninstall_repl_displayhook():
 """
 Uninstall the matplotlib display hook.

 .. warning::

 Need IPython >= 2 for this to work. For IPython < 2 will raise a
 ``NotImplementedError``

 .. warning::

 If you are using vanilla python and have installed another
 display hook this will reset ``sys.displayhook`` to what ever
 function was there when matplotlib installed it's displayhook,
 possibly discarding your changes.
 """
 global _IP_REGISTERED
 global _INSTALL FIG_OBSERVER

```

```

if _IP_REGISTERED:
 from IPython import get_ipython
 ip = get_ipython()
 try:
 ip.events.unregister('post_execute', _IP_REGISTERED)
 except AttributeError as err:
 raise NotImplementedError("Can not unregister events "
 "in IPython < 2.0") from err
 _IP_REGISTERED = None

if _INSTALL FIG_OBSERVER:
 _INSTALL FIG_OBSERVER = False

draw_all = _pylab_helpers.Gcf.draw_all

@functools.wraps(matplotlib.set_loglevel)
def set_loglevel(*args, **kwargs): # Ensure this appears in the pyplot docs.
 return matplotlib.set_loglevel(*args, **kwargs)

 @_copy_docstring_and_deprecators(Artist.findobj)
def findobj(o=None, match=None, include_self=True):
 if o is None:
 o = gcf()
 return o.findobj(match, include_self=include_self)

def _get_required_interactive_framework(backend_mod):
 return getattr(
 backend_mod.FigureCanvas, "required_interactive_framework", None)

def switch_backend(newbackend):
 """
 Close all open figures and set the Matplotlib backend.

 The argument is case-insensitive. Switching to an interactive backend is
 possible only if no event loop for another interactive backend has started.
 Switching to and from non-interactive backends is always possible.

 Parameters

 newbackend : str
 The name of the backend to use.
 """
 global _backend_mod

 close("all")

 if newbackend is rcsetup._auto_backend_sentinel:
 # Don't try to fallback on the cairo-based backends as they each have
 # an additional dependency (pycairo) over the agg-based backend, and
 # are of worse quality.
 for candidate in [
 "macosx", "qt5agg", "qt4agg", "gtk3agg", "tkagg", "wxagg"]:
 try:
 switch_backend(candidate)
 except ImportError:
 continue
 else:
 rcParamsOrig['backend'] = candidate

```

```

 return
 else:
 # Switching to Agg should always succeed; if it doesn't, let the
 # exception propagate out.
 switch_backend("agg")
 rcParamsOrig["backend"] = "agg"
 return

Backends are implemented as modules, but "inherit" default method
implementations from backend_bases._Backend. This is achieved by
creating a "class" that inherits from backend_bases._Backend and whose
body is filled with the module's globals.

backend_name = cbook._backend_module_name(newbackend)

class backend_mod(matplotlib.backend_bases._Backend):
 locals().update(vars(importlib.import_module(backend_name)))

required_framework = _get_required_interactive_framework(backend_mod)
if required_framework is not None:
 current_framework = cbook._get_running_interactive_framework()
 if (current_framework and required_framework
 and current_framework != required_framework):
 raise ImportError(
 "Cannot load backend {!r} which requires the {!r} interactive "
 "framework, as {!r} is currently running".format(
 newbackend, required_framework, current_framework))

 _log.debug("Loaded backend %s version %s.",
 newbackend, backend_mod.backend_version)

rcParams['backend'] = rcParamsDefault['backend'] = newbackend
_backend_mod = backend_mod
for func_name in ["new_figure_manager", "draw_if_interactive", "show"]:
 globals()[func_name].__signature__ = inspect.signature(
 getattr(backend_mod, func_name))

Need to keep a global reference to the backend for compatibility reasons.
See https://github.com/matplotlib/matplotlib/issues/6092
matplotlib.backends.backend = newbackend

def _warn_if_gui_out_of_main_thread():
 if (_get_required_interactive_framework(_backend_mod)
 and threading.current_thread() is not threading.main_thread()):
 cbook._warn_external(
 "Starting a Matplotlib GUI outside of the main thread will likely "
 "fail.")

This function's signature is rewritten upon backend-load by switch_backend.
def new_figure_manager(*args, **kwargs):
 """Create a new figure manager instance."""
 _warn_if_gui_out_of_main_thread()
 return _backend_mod.new_figure_manager(*args, **kwargs)

This function's signature is rewritten upon backend-load by switch_backend.
def draw_if_interactive(*args, **kwargs):
 return _backend_mod.draw_if_interactive(*args, **kwargs)

This function's signature is rewritten upon backend-load by switch_backend.

```

```

def show(*args, **kwargs):
 """
 Display all figures.

 When running in ipython with its pylab mode, display all
 figures and return to the ipython prompt.

 In non-interactive mode, display all figures and block until
 the figures have been closed; in interactive mode it has no
 effect unless figures were created prior to a change from
 non-interactive to interactive mode (not recommended). In
 that case it displays the figures but does not block.

 Parameters

 block : bool, optional
 This is experimental, and may be set to ``True`` or ``False`` to
 override the blocking behavior described above.
 """
 _warn_if_gui_out_of_main_thread()
 return _backend_mod.show(*args, **kwargs)

def isinteractive():
 """Return whether to redraw after every plotting command."""
 return matplotlib.is_interactive()

def ioff():
 """Turn the interactive mode off."""
 matplotlib.interactive(False)
 uninstall_repl_displayhook()

def ion():
 """Turn the interactive mode on."""
 matplotlib.interactive(True)
 install_repl_displayhook()

def pause(interval):
 """
 Pause for *interval* seconds.

 If there is an active figure, it will be updated and displayed before the
 pause, and the GUI event loop (if any) will run during the pause.

 This can be used for crude animation. For more complex animation, see
 :mod:`matplotlib.animation`.
 """

Notes

This function is experimental; its behavior may be changed or extended in a
future release.
"""
manager = _pylab_helpers.Gcf.get_active()
if manager is not None:
 canvas = manager.canvas
 if canvas.figure.stale:
 canvas.draw_idle()
 show(block=False)
 canvas.start_event_loop(interval)
else:

```

```
time.sleep(interval)

 @_copy_docstring_and_deprecators(matplotlib.rc)
def rc(group, **kwargs):
 matplotlib.rc(group, **kwargs)

 @_copy_docstring_and_deprecators(matplotlib.rc_context)
def rc_context(rc=None, fname=None):
 return matplotlib.rc_context(rc, fname)

 @_copy_docstring_and_deprecators(matplotlib.rcdefaults)
def rcdefaults():
 matplotlib.rcdefaults()
 if matplotlib.is_interactive():
 draw_all()

getp/get/setp are explicitly reexported so that they show up in pyplot docs.

 @_copy_docstring_and_deprecators(matplotlib.artist.getp)
def getp(obj, *args, **kwargs):
 return matplotlib.artist.getp(obj, *args, **kwargs)

 @_copy_docstring_and_deprecators(matplotlib.artist.get)
def get(obj, *args, **kwargs):
 return matplotlib.artist.get(obj, *args, **kwargs)

 @_copy_docstring_and_deprecators(matplotlib.artist.setp)
def setp(obj, *args, **kwargs):
 return matplotlib.artist.setp(obj, *args, **kwargs)

def xkcd(scale=1, length=100, randomness=2):
 """
 Turn on `xkcd <https://xkcd.com/>`_ sketch-style drawing mode. This will
 only have effect on things drawn after this function is called.

 For best results, the "Humor Sans" font should be installed: it is
 not included with Matplotlib.

 Parameters

 scale : float, optional
 The amplitude of the wiggle perpendicular to the source line.
 length : float, optional
 The length of the wiggle along the line.
 randomness : float, optional
 The scale factor by which the length is shrunken or expanded.

 Notes

 This function works by a number of rcParams, so it will probably
 override others you have set before.

 If you want the effects of this function to be temporary, it can
 be used as a context manager, for example::
```

```

with plt.xkcd():
 # This figure will be in XKCD-style
 fig1 = plt.figure()
 # ...

 # This figure will be in regular style
 fig2 = plt.figure()
"""
return _xkcd(scale, length, randomness)

class _xkcd:
 # This cannot be implemented in terms of rc_context() because this needs to
 # work as a non-contextmanager too.

 def __init__(self, scale, length, randomness):
 self._orig = rcParams.copy()

 if rcParams['text.usetex']:
 raise RuntimeError(
 "xkcd mode is not compatible with text.usetex = True")

 from matplotlib import patheffects
 rcParams.update({
 'font.family': ['xkcd', 'xkcd Script', 'Humor Sans', 'Comic Neue',
 'Comic Sans MS'],
 'font.size': 14.0,
 'path.sketch': (scale, length, randomness),
 'path.effects': [
 patheffects.withStroke(linewidth=4, foreground="w")],
 'axes.linewidth': 1.5,
 'lines.linewidth': 2.0,
 'figure.facecolor': 'white',
 'grid.linewidth': 0.0,
 'axes.grid': False,
 'axes.unicode_minus': False,
 'axes.edgecolor': 'black',
 'xtick.major.size': 8,
 'xtick.major.width': 3,
 'ytick.major.size': 8,
 'ytick.major.width': 3,
 })
)

 def __enter__(self):
 return self

 def __exit__(self, *args):
 dict.update(rcParams, self._orig)

Figures

def figure(num=None, # autoincrement if None, else integer from 1-N
 figsize=None, # defaults to rc figure.figsize
 dpi=None, # defaults to rc figure.dpi
 facecolor=None, # defaults to rc figure.facecolor
 edgecolor=None, # defaults to rc figure.edgecolor
 frameon=True,
 FigureClass=Figure,
 clear=False,
 **kwargs
):
"""

```

```
Create a new figure, or activate an existing figure.

Parameters

num : int or str, optional
 A unique identifier for the figure.

 If a figure with that identifier already exists, this figure is made
 active and returned. An integer refers to the ``Figure.number`` attribute,
 a string refers to the figure label.

 If there is no figure with the identifier or *num* is not given, a new
 figure is created, made active and returned. If *num* is an int, it
 will be used for the ``Figure.number`` attribute, otherwise, an
 auto-generated integer value is used (starting at 1 and incremented
 for each new figure). If *num* is a string, the figure label and the
 window title is set to this value.

figsize : (float, float), default: :rc:`figure.figsize`
 Width, height in inches.

dpi : float, default: :rc:`figure.dpi`
 The resolution of the figure in dots-per-inch.

facecolor : color, default: :rc:`figure.facecolor`
 The background color.

edgecolor : color, default: :rc:`figure.edgecolor`
 The border color.

frameon : bool, default: True
 If False, suppress drawing the figure frame.

FigureClass : subclass of `~matplotlib.figure.Figure`
 Optionally use a custom `Figure` instance.

clear : bool, default: False
 If True and the figure already exists, then it is cleared.

Returns

`~matplotlib.figure.Figure`
 The `Figure` instance returned will also be passed to
 new_figure_manager in the backends, which allows to hook custom
 `Figure` classes into the pyplot interface. Additional kwargs will be
 passed to the `Figure` init function.

Notes

If you are creating many figures, make sure you explicitly call
`pyplot.close` on the figures you are not using, because this will
enable pyplot to properly clean up the memory.

`~matplotlib.rcParams` defines the default values, which can be modified
in the matplotlibrc file.

"""

if figsize is None:
 figsize = rcParams['figure.figsize']
if dpi is None:
 dpi = rcParams['figure.dpi']
if facecolor is None:
 facecolor = rcParams['figure.facecolor']
```

```

if edgecolor is None:
 edgecolor = rcParams['figure.edgecolor']

allnums = get_fignums()
next_num = max(allnums) + 1 if allnums else 1
figLabel = ''
if num is None:
 num = next_num
elif isinstance(num, str):
 figLabel = num
 allLabels = get_figlabels()
 if figLabel not in allLabels:
 if figLabel == 'all':
 cbook._warn_external(
 "close('all') closes all existing figures")
 num = next_num
 else:
 inum = allLabels.index(figLabel)
 num = allnums[inum]
else:
 num = int(num) # crude validation of num argument

figManager = _pylab_helpers.Gcf.get_fig_manager(num)
if figManager is None:
 max_open_warning = rcParams['figure.max_open_warning']

 if len(allnums) == max_open_warning >= 1:
 cbook._warn_external(
 "More than %d figures have been opened. Figures "
 "created through the pyplot interface "
 "(`matplotlib.pyplot.figure`) are retained until "
 "explicitly closed and may consume too much memory. "
 "(To control this warning, see the rcParam "
 "`figure.max_open_warning`)." %
 max_open_warning, RuntimeWarning)

 if get_backend().lower() == 'ps':
 dpi = 72

 figManager = new_figure_manager(num, figsize=figsize,
 dpi=dpi,
 facecolor=facecolor,
 edgecolor=edgecolor,
 frameon=frameon,
 FigureClass=FigureClass,
 **kwargs)
 fig = figManager.canvas.figure
 if figLabel:
 fig.set_label(figLabel)

 _pylab_helpers.Gcf._set_new_active_manager(figManager)

make sure backends (inline) that we don't ship that expect this
to be called in plotting commands to make the figure call show
still work. There is probably a better way to do this in the
FigureManager base class.
draw_if_interactive()

if _INSTALL FIG_OBSERVER:
 fig.stale_callback = _auto_draw_if_interactive

if clear:
 figManager.canvas.figure.clear()

```

```

 return figManager.canvas.figure

def _auto_draw_if_interactive(fig, val):
 """
 An internal helper function for making sure that auto-redrawing
 works as intended in the plain python repl.

 Parameters

 fig : Figure
 A figure object which is assumed to be associated with a canvas
 """
 if (val and matplotlib.is_interactive()
 and not fig.canvas.is_saving()
 and not fig.canvas._is_idle_drawing):
 # Some artists can mark themselves as stale in the middle of drawing
 # (e.g. axes position & tick labels being computed at draw time), but
 # this shouldn't trigger a redraw because the current redraw will
 # already take them into account.
 with fig.canvas._idle_draw_ctx():
 fig.canvas.draw_idle()

def gcf():
 """
 Get the current figure.

 If no current figure exists, a new one is created using
 `~.pyplot.figure()`.

 """
 figManager = _pylab_helpers.Gcf.get_active()
 if figManager is not None:
 return figManager.canvas.figure
 else:
 return figure()

def fignum_exists(num):
 """
 Return whether the figure with the given id exists."""
 return _pylab_helpers.Gcf.has_fignum(num) or num in get_figlabels()

def get_fignums():
 """
 Return a list of existing figure numbers."""
 return sorted(_pylab_helpers.Gcf.figs)

def get_figlabels():
 """
 Return a list of existing figure labels."""
 figManagers = _pylab_helpers.Gcf.get_all_fig_managers()
 figManagers.sort(key=lambda m: m.num)
 return [m.canvas.figure.get_label() for m in figManagers]

def get_current_fig_manager():
 """
 Return the figure manager of the current figure.

 The figure manager is a container for the actual backend-depended window
 that displays the figure on screen.

```



```

"""Clear the current figure."""
gcf().clf()

def draw():
 """
 Redraw the current figure.

 This is used to update a figure that has been altered, but not
 automatically re-drawn. If interactive mode is on (via `ion()`), this
 should be only rarely needed, but there may be ways to modify the state of
 a figure without marking it as "stale". Please report these cases as bugs.

 This is equivalent to calling ``fig.canvas.draw_idle()`` , where ``fig`` is
 the current figure.
 """
 gcf().canvas.draw_idle()

 @_copy_docstring_and_deprecators(Figure.savefig)
def savefig(*args, **kwargs):
 fig = gcf()
 res = fig.savefig(*args, **kwargs)
 fig.canvas.draw_idle() # need this if 'transparent=True' to reset colors
 return res

Putting things in figures

def figlegend(*args, **kwargs):
 return gcf().legend(*args, **kwargs)
if Figure.legend.__doc__:
 figlegend.__doc__ = Figure.legend.__doc__.replace("legend()", "figlegend()")

Axes

@docstring.dedent_interpd
def axes(arg=None, **kwargs):
 """
 Add an axes to the current figure and make it the current axes.

 Call signatures::

 plt.axes()
 plt.axes(rect, projection=None, polar=False, **kwargs)
 plt.axes(ax)

 Parameters

 arg : None or 4-tuple
 The exact behavior of this function depends on the type:
 - *None*: A new full window axes is added using
          ```subplot(111, **kwargs)```
        - 4-tuple of floats *rect* = ``[left, bottom, width, height]``.
          A new axes is added with dimensions *rect* in normalized
          (0, 1) units using `~.Figure.add_axes` on the current figure.

    projection : {None, 'aitoff', 'hammer', 'lambert', 'mollweide', \
    'polar', 'rectilinear', str}, optional
        The projection type of the `~.axes.Axes` . *str* is the name of

```

```
a custom projection, see `~matplotlib.projections`. The default
None results in a 'rectilinear' projection.

polar : bool, default: False
    If True, equivalent to projection='polar'.

sharex, sharey : `~.axes.Axes`, optional
    Share the x or y `~matplotlib.axis` with sharex and/or sharey.
    The axis will have the same limits, ticks, and scale as the axis
    of the shared axes.

label : str
    A label for the returned axes.

>Returns
-----
`~.axes.Axes` (or a subclass of `~.axes.Axes`)
    The returned axes class depends on the projection used. It is
    `~.axes.Axes` if rectilinear projection are used and
    `~.projections.polar.PolarAxes` if polar projection
    are used.

Other Parameters
-----
**kwargs
    This method also takes the keyword arguments for
    the returned axes class. The keyword arguments for the
    rectilinear axes class `~.axes.Axes` can be found in
    the following table but there might also be other keyword
    arguments if another projection is used, see the actual axes
    class.

%(Axes)s

Notes
-----
If the figure already has a axes with key (*args*, *kwargs*) then it will simply make that axes current and
return it. This behavior is deprecated. Meanwhile, if you do
not want this behavior (i.e., you want to force the creation of a
new axes), you must use a unique set of args and kwargs. The axes
*label* attribute has been exposed for this purpose: if you want
two axes that are otherwise identical to be added to the figure,
make sure you give them unique labels.

See Also
-----
.Figure.add_axes
.pyplot.subplot
.Figure.add_subplot
.Figure.subplots
.pyplot.subplots

Examples
-----
::

    # Creating a new full window axes
    plt.axes()

    # Creating a new axes with specified dimensions and some kwargs
    plt.axes((left, bottom, width, height), facecolor='w')

    ...

```

```

if arg is None:
    return subplot(111, **kwargs)
else:
    return gcf().add_axes(arg, **kwargs)

def delaxes(ax=None):
    """
    Remove an `~.axes.Axes` (defaulting to the current axes) from its figure.
    """
    if ax is None:
        ax = gca()
    ax.remove()

def sca(ax):
    """
    Set the current Axes to *ax* and the current Figure to the parent of *ax*.
    """
    if not hasattr(ax.figure.canvas, "manager"):
        raise ValueError("Axes parent figure is not managed by pyplot")
    _pylab_helpers.Gcf.set_active(ax.figure.canvas.manager)
    ax.figure.sca(ax)

## More ways of creating axes ##

@docstring.dedent_interpd
def subplot(*args, **kwargs):
    """
    Add a subplot to the current figure.

    Wrapper of `~.Figure.add_subplot` with a difference in behavior
    explained in the notes section.

    Call signatures::

        subplot(nrows, ncols, index, **kwargs)
        subplot(pos, **kwargs)
        subplot(**kwargs)
        subplot(ax)

    Parameters
    -----
    *args, default: (1, 1, 1)
        Either a 3-digit integer or three separate integers
        describing the position of the subplot. If the three
        integers are *nrows*, *ncols*, and *index* in order, the
        subplot will take the *index* position on a grid with *nrows*
        rows and *ncols* columns. *index* starts at 1 in the upper left
        corner and increases to the right.

        *pos* is a three digit integer, where the first digit is the
        number of rows, the second the number of columns, and the third
        the index of the subplot. i.e. fig.add_subplot(235) is the same as
        fig.add_subplot(2, 3, 5). Note that all integers must be less than
        10 for this form to work.

        projection : {None, 'aitoff', 'hammer', 'lambert', 'mollweide', \
'polar', 'rectilinear', str}, optional
            The projection type of the subplot (`~.axes.Axes`). *str* is the name
            of a custom projection, see `~matplotlib.projections`. The default

```

```
None results in a 'rectilinear' projection.

polar : bool, default: False
    If True, equivalent to projection='polar'.

sharex, sharey : `~.axes.Axes`, optional
    Share the x or y `~matplotlib.axis` with sharex and/or sharey. The
    axis will have the same limits, ticks, and scale as the axis of the
    shared axes.

label : str
    A label for the returned axes.

Returns
-----
an `~.axes.SubplotBase` subclass of `~.axes.Axes` (or a subclass of \
`~.axes.Axes`)

The axes of the subplot. The returned axes base class depends on
the projection used. It is `~.axes.Axes` if rectilinear projection
are used and `~.projections.polar.PolarAxes` if polar projection
are used. The returned axes is then a subplot subclass of the
base class.

Other Parameters
-----
**kwargs
    This method also takes the keyword arguments for the returned axes
    base class; except for the *figure* argument. The keyword arguments
    for the rectilinear base class `~.axes.Axes` can be found in
    the following table but there might also be other keyword
    arguments if another projection is used.

%(Axes)s

Notes
-----
Creating a subplot will delete any pre-existing subplot that overlaps
with it beyond sharing a boundary:::

import matplotlib.pyplot as plt
# plot a line, implicitly creating a subplot(111)
plt.plot([1, 2, 3])
# now create a subplot which represents the top plot of a grid
# with 2 rows and 1 column. Since this subplot will overlap the
# first, the plot (and its axes) previously created, will be removed
plt.subplot(211)

If you do not want this behavior, use the `Figure.add_subplot` method
or the `~.pyplot.axes` function instead.

If the figure already has a subplot with key (*args*, *kwargs*) then it will simply make that subplot current and
return it. This behavior is deprecated. Meanwhile, if you do
not want this behavior (i.e., you want to force the creation of a
new subplot), you must use a unique set of args and kwargs. The axes
*label* attribute has been exposed for this purpose: if you want
two subplots that are otherwise identical to be added to the figure,
make sure you give them unique labels.

In rare circumstances, `~.add_subplot` may be called with a single
argument, a subplot axes instance already created in the
present figure but not in the figure's list of axes.
```

```
See Also
-----
.Figure.add_subplot
.pyplot.subplots
.pyplot.axes
.Figure.subplots

Examples
-----
::

    plt.subplot(221)

    # equivalent but more general
    ax1=plt.subplot(2, 2, 1)

    # add a subplot with no frame
    ax2=plt.subplot(222, frameon=False)

    # add a polar subplot
    plt.subplot(223, projection='polar')

    # add a red subplot that shares the x-axis with ax1
    plt.subplot(224, sharex=ax1, facecolor='red')

    # delete ax2 from the figure
    plt.delaxes(ax2)

    # add ax2 to the figure again
    plt.subplot(ax2)
"""

# if subplot called without arguments, create subplot(1, 1, 1)
if len(args) == 0:
    args = (1, 1, 1)

# This check was added because it is very easy to type
# subplot(1, 2, False) when subplots(1, 2, False) was intended
# (sharex=False, that is). In most cases, no error will
# ever occur, but mysterious behavior can result because what was
# intended to be the sharex argument is instead treated as a
# subplot index for subplot()
if len(args) >= 3 and isinstance(args[2], bool):
    cbook._warn_external("The subplot index argument to subplot() appears "
                         "to be a boolean. Did you intend to use "
                         "'subplots()'?")

# Check for nrows and ncols, which are not valid subplot args:
if 'nrows' in kwargs or 'ncols' in kwargs:
    raise TypeError("subplot() got an unexpected keyword argument 'ncols' "
                    "and/or 'nrows'. Did you intend to call subplots()?")


fig = gcf()
ax = fig.add_subplot(*args, **kwargs)
bbox = ax.bbox
axes_to_delete = []
for other_ax in fig.axes:
    if other_ax == ax:
        continue
    if bbox.fully_overlaps(other_ax.bbox):
        axes_to_delete.append(other_ax)
for ax_to_del in axes_to_delete:
    delaxes(ax_to_del)
```

```

    return ax

def subplots(nrows=1, ncols=1, sharex=False, sharey=False, squeeze=True,
            subplot_kw=None, gridspec_kw=None, **fig_kw):
    """
    Create a figure and a set of subplots.

    This utility wrapper makes it convenient to create common layouts of
    subplots, including the enclosing figure object, in a single call.

    Parameters
    -----
    nrows, ncols : int, default: 1
        Number of rows/columns of the subplot grid.

    sharex, sharey : bool or {'none', 'all', 'row', 'col'}, default: False
        Controls sharing of properties among x (*sharex*) or y (*sharey*)
        axes:

        - True or 'all': x- or y-axis will be shared among all subplots.
        - False or 'none': each subplot x- or y-axis will be independent.
        - 'row': each subplot row will share an x- or y-axis.
        - 'col': each subplot column will share an x- or y-axis.

        When subplots have a shared x-axis along a column, only the x tick
        labels of the bottom subplot are created. Similarly, when subplots
        have a shared y-axis along a row, only the y tick labels of the first
        column subplot are created. To later turn other subplots' ticklabels
        on, use `~matplotlib.axes.Axes.tick_params`.

    squeeze : bool, default: True
        - If True, extra dimensions are squeezed out from the returned
          array of `~matplotlib.axes.Axes`:

            - if only one subplot is constructed (nrows=ncols=1), the
              resulting single Axes object is returned as a scalar.
            - for Nx1 or 1xM subplots, the returned object is a 1D numpy
              object array of Axes objects.
            - for NxM, subplots with N>1 and M>1 are returned as a 2D array.

        - If False, no squeezing at all is done: the returned Axes object is
          always a 2D array containing Axes instances, even if it ends up
          being 1x1.

    subplot_kw : dict, optional
        Dict with keywords passed to the
        `~matplotlib.figure.Figure.add_subplot` call used to create each
        subplot.

    gridspec_kw : dict, optional
        Dict with keywords passed to the `~matplotlib.gridspec.GridSpec`
        constructor used to create the grid the subplots are placed on.

    **fig_kw
        All additional keyword arguments are passed to the
        `~pyplot.figure` call.

    Returns
    -----
    fig : `~.figure.Figure`
```

```
ax : `~.axes.Axes` or array of Axes
    *ax* can be either a single `~matplotlib.axes.Axes` object or an
    array of Axes objects if more than one subplot was created. The
    dimensions of the resulting array can be controlled with the squeeze
    keyword, see above.
```

Typical idioms for handling the return value are::

```
# using the variable ax for single a Axes
fig, ax = plt.subplots()

# using the variable axs for multiple Axes
fig, axs = plt.subplots(2, 2)

# using tuple unpacking for multiple Axes
fig, (ax1, ax2) = plt.subplot(1, 2)
fig, ((ax1, ax2), (ax3, ax4)) = plt.subplot(2, 2)
```

The names ``ax`` and pluralized ``axs`` are preferred over ``axes`` because for the latter it's not clear if it refers to a single `~.axes.Axes` instance or a collection of these.

See Also

- .pyplot.figure
- .pyplot.subplot
- .pyplot.axes
- .Figure.subplots
- .Figure.add_subplot

Examples

::

```
# First create some toy data:
x = np.linspace(0, 2*np.pi, 400)
y = np.sin(x**2)

# Create just a figure and only one subplot
fig, ax = plt.subplots()
ax.plot(x, y)
ax.set_title('Simple plot')

# Create two subplots and unpack the output array immediately
f, (ax1, ax2) = plt.subplots(1, 2, sharey=True)
ax1.plot(x, y)
ax1.set_title('Sharing Y axis')
ax2.scatter(x, y)

# Create four polar axes and access them through the returned array
fig, axs = plt.subplots(2, 2, subplot_kw=dict(polar=True))
axs[0, 0].plot(x, y)
axs[1, 1].scatter(x, y)

# Share a X axis with each column of subplots
plt.subplots(2, 2, sharex='col')

# Share a Y axis with each row of subplots
plt.subplots(2, 2, sharey='row')

# Share both X and Y axes with all subplots
plt.subplots(2, 2, sharex='all', sharey='all')
```

```

# Note that this is the same as
plt.subplots(2, 2, sharex=True, sharey=True)

# Create figure number 10 with a single subplot
# and clears it if it already exists.
fig, ax = plt.subplots(num=10, clear=True)

"""

fig = figure(**fig_kw)
axs = fig.subplots(nrows=nrows, ncols=ncols, sharex=sharex, sharey=sharey,
                   squeeze=squeeze, subplot_kw=subplot_kw,
                   gridspec_kw=gridspec_kw)
return fig, axs

def subplot2grid(shape, loc, rowspan=1, colspan=1, fig=None, **kwargs):
    """
    Create an axis at specific location inside a regular grid.

    Parameters
    -----
    shape : (int, int)
        Number of rows and of columns of the grid in which to place axis.
    loc : (int, int)
        Row number and column number of the axis location within the grid.
    rowspan : int
        Number of rows for the axis to span to the right.
    colspan : int
        Number of columns for the axis to span downwards.
    fig : `.Figure`, optional
        Figure to place axis in. Defaults to current figure.
    **kwargs
        Additional keyword arguments are handed to `add_subplot`.

    Notes
    -----
    The following call ::

        subplot2grid(shape, loc, rowspan=1, colspan=1)

    is identical to ::

        gridspec = GridSpec(shape[0], shape[1])
        subplotspec = gridspec.new_subplotspec(loc, rowspan, colspan)
        subplot(subplotspec)

    """

    if fig is None:
        fig = gcf()

    s1, s2 = shape
    subplotspec = GridSpec(s1, s2).new_subplotspec(loc,
                                                   rowspan=rowspan,
                                                   colspan=colspan)

    ax = fig.add_subplot(subplotspec, **kwargs)
    bbox = ax.bbox
    axes_to_delete = []
    for other_ax in fig.axes:
        if other_ax == ax:
            continue
        if bbox.fully_overlaps(other_ax.bbox):
            axes_to_delete.append(other_ax)
    for ax_to_del in axes_to_delete:

```

```

    delaxes(ax_to_del)

    return ax

def twinx(ax=None):
    """
    Make and return a second axes that shares the *x*-axis. The new axes will
    overlay *ax* (or the current axes if *ax* is *None*), and its ticks will be
    on the right.

    Examples
    -----
    :doc:`/gallery/subplots_axes_and_figures/two_scales`
    """
    if ax is None:
        ax = gca()
    ax1 = ax.twinx()
    return ax1

def twiny(ax=None):
    """
    Make and return a second axes that shares the *y*-axis. The new axes will
    overlay *ax* (or the current axes if *ax* is *None*), and its ticks will be
    on the top.

    Examples
    -----
    :doc:`/gallery/subplots_axes_and_figures/two_scales`
    """
    if ax is None:
        ax = gca()
    ax1 = ax.twiny()
    return ax1

def subplot_tool(targetfig=None):
    """
    Launch a subplot tool window for a figure.

    A :class:`matplotlib.widgets.SubplotTool` instance is returned.

    """
    if targetfig is None:
        targetfig = gcf()

    with rc_context({'toolbar': 'None'}): # No nav toolbar for the toolfig.
        toolfig = figure(figsize=(6, 3))
        toolfig.subplots_adjust(top=0.9)

    if hasattr(targetfig.canvas, "manager"): # Restore the current figure.
        _pylab_helpers.Gcf.set_active(targetfig.canvas.manager)

    return SubplotTool(targetfig, toolfig)

# After deprecation elapses, this can be autogenerated by boilerplate.py.
@cbook._make_keyword_only("3.3", "pad")
def tight_layout(pad=1.08, h_pad=None, w_pad=None, rect=None):
    """
    Adjust the padding between and around subplots.

    Parameters

```

```
-----  
pad : float, default: 1.08  
    Padding between the figure edge and the edges of subplots,  
    as a fraction of the font size.  
h_pad, w_pad : float, default: *pad*  
    Padding (height/width) between edges of adjacent subplots,  
    as a fraction of the font size.  
rect : tuple (left, bottom, right, top), default: (0, 0, 1, 1)  
    A rectangle in normalized figure coordinates into which the whole  
    subplots area (including labels) will fit.  
"""  
gcf().tight_layout(pad=pad, h_pad=h_pad, w_pad=w_pad, rect=rect)  
  
def box(on=None):  
    """  
    Turn the axes box on or off on the current axes.  
  
    Parameters  
    -----  
    on : bool or None  
        The new `~matplotlib.axes.Axes` box state. If ``None``, toggle  
        the state.  
  
    See Also  
    -----  
    :meth:`matplotlib.axes.Axes.set_frame_on`  
    :meth:`matplotlib.axes.Axes.get_frame_on`  
    """  
    ax = gca()  
    if on is None:  
        on = not ax.get_frame_on()  
    ax.set_frame_on(on)  
  
## Axis ##  
  
def xlim(*args, **kwargs):  
    """  
    Get or set the x limits of the current axes.  
  
    Call signatures::  
  
        left, right = xlim()    # return the current xlim  
        xlim((left, right))    # set the xlim to left, right  
        xlim(left, right)      # set the xlim to left, right  
  
    If you do not specify args, you can pass *left* or *right* as kwargs,  
    i.e.:  
  
        xlim(right=3)    # adjust the right leaving left unchanged  
        xlim(left=1)     # adjust the left leaving right unchanged  
  
    Setting limits turns autoscaling off for the x-axis.  
  
    Returns  
    -----  
    left, right  
        A tuple of the new x-axis limits.  
  
    Notes  
    -----  
    Calling this function with no arguments (e.g. ``xlim()``) is the pyplot
```

```

equivalent of calling `~.Axes.get_xlim` on the current axes.
Calling this function with arguments is the pyplot equivalent of calling
`~.Axes.set_xlim` on the current axes. All arguments are passed though.
"""
ax = gca()
if not args and not kwargs:
    return ax.get_xlim()
ret = ax.set_xlim(*args, **kwargs)
return ret

def ylim(*args, **kwargs):
    """
    Get or set the y-limits of the current axes.

    Call signatures::

        bottom, top = ylim()    # return the current ylim
        ylim((bottom, top))    # set the ylim to bottom, top
        ylim(bottom, top)      # set the ylim to bottom, top

    If you do not specify args, you can alternatively pass *bottom* or
    *top* as kwargs, i.e.::

        ylim(top=3)    # adjust the top leaving bottom unchanged
        ylim(bottom=1) # adjust the bottom leaving top unchanged

    Setting limits turns autoscaling off for the y-axis.

    Returns
    ------
    bottom, top
        A tuple of the new y-axis limits.

    Notes
    -----
    Calling this function with no arguments (e.g. ``ylim()``) is the pyplot
    equivalent of calling `~.Axes.get_ylim` on the current axes.
    Calling this function with arguments is the pyplot equivalent of calling
    `~.Axes.set_ylim` on the current axes. All arguments are passed though.
    """
    ax = gca()
    if not args and not kwargs:
        return ax.get_ylim()
    ret = ax.set_ylim(*args, **kwargs)
    return ret

def xticks(ticks=None, labels=None, **kwargs):
    """
    Get or set the current tick locations and labels of the x-axis.

    Pass no arguments to return the current values without modifying them.

    Parameters
    -----
    ticks : array-like, optional
        The list of xtick locations. Passing an empty list removes all xticks.
    labels : array-like, optional
        The labels to place at the given *ticks* locations. This argument can
        only be passed if *ticks* is passed as well.
    **kwargs
        `.Text` properties can be used to control the appearance of the labels.

```

Returns

locs

The list of xtick locations.

labels

The list of xlabel `Text` objects.

Notes

Calling this function with no arguments (e.g. ``xticks()``) is the pyplot equivalent of calling `~.Axes.get_xticks` and `~.Axes.get_xticklabels` on the current axes.

Calling this function with arguments is the pyplot equivalent of calling `~.Axes.set_xticks` and `~.Axes.set_xticklabels` on the current axes.

Examples

```
>>> locs, labels = xticks() # Get the current locations and labels.
>>> xticks(np.arange(0, 1, step=0.2)) # Set label locations.
>>> xticks(np.arange(3), ['Tom', 'Dick', 'Sue']) # Set text labels.
>>> xticks([0, 1, 2], ['January', 'February', 'March'],
...         rotation=20) # Set text labels and properties.
>>> xticks([]) # Disable xticks.
"""
ax = gca()

if ticks is None:
    locs = ax.get_xticks()
    if labels is not None:
        raise TypeError("xticks(): Parameter 'labels' can't be set "
                        "without setting 'ticks'")
else:
    locs = ax.set_xticks(ticks)

if labels is None:
    labels = ax.get_xticklabels()
else:
    labels = ax.set_xticklabels(labels, **kwargs)
for l in labels:
    l.update(kwargs)

return locs, labels

def yticks(ticks=None, labels=None, **kwargs):
"""
Get or set the current tick locations and labels of the y-axis.

Pass no arguments to return the current values without modifying them.

Parameters
-----
ticks : array-like, optional
    The list of xtick locations. Passing an empty list removes all xticks.
labels : array-like, optional
    The labels to place at the given *ticks* locations. This argument can
    only be passed if *ticks* is passed as well.
**kwargs
    `Text` properties can be used to control the appearance of the labels.

Returns
-----
```

```

locs
    The list of ytick locations.
labels
    The list of ylabel `Text` objects.

Notes
-----
Calling this function with no arguments (e.g. ``yticks()``) is the pyplot
equivalent of calling `~.Axes.get_yticks` and `~.Axes.get_yticklabels` on
the current axes.
Calling this function with arguments is the pyplot equivalent of calling
`~.Axes.set_yticks` and `~.Axes.set_yticklabels` on the current axes.

Examples
-----
>>> locs, labels = yticks() # Get the current locations and labels.
>>> yticks(np.arange(0, 1, step=0.2)) # Set label locations.
>>> yticks(np.arange(3), ['Tom', 'Dick', 'Sue']) # Set text labels.
>>> yticks([0, 1, 2], ['January', 'February', 'March'],
...           rotation=45) # Set text labels and properties.
>>> yticks([]) # Disable yticks.
"""
ax = gca()

if ticks is None:
    locs = ax.get_yticks()
    if labels is not None:
        raise TypeError("yticks(): Parameter 'labels' can't be set "
                        "without setting 'ticks'")
else:
    locs = ax.set_yticks(ticks)

if labels is None:
    labels = ax.get_yticklabels()
else:
    labels = ax.set_yticklabels(labels, **kwargs)
for l in labels:
    l.update(kwargs)

return locs, labels

def rgrids(*args, **kwargs):
    """
    Get or set the radial gridlines on the current polar plot.

    Call signatures::

        lines, labels = rgrids()
        lines, labels = rgrids(radii, labels=None, angle=22.5, fmt=None, **kwargs)

    When called with no arguments, `rgrids` simply returns the tuple
    (*lines*, *labels*). When called with arguments, the labels will
    appear at the specified radial distances and angle.

    Parameters
    -----
    radii : tuple with floats
        The radii for the radial gridlines

    labels : tuple with strings or None
        The labels to use at each radial gridline. The
        `matplotlib.ticker.ScalarFormatter` will be used if None.

```

```
angle : float
    The angular position of the radius labels in degrees.

fmt : str or None
    Format string used in `matplotlib.ticker.FormatStrFormatter`.
    For example '%f'.

Returns
-----
lines : list of `~.lines.Line2D`
    The radial gridlines.

labels : list of `~.text.Text`
    The tick labels.

Other Parameters
-----
**kwargs
    *kwargs* are optional `~.Text` properties for the labels.

See Also
-----
.pyplot.thetagrids
.projections.polar.PolarAxes.set_rgrids
.Axis.get_gridlines
.Axis.get_ticklabels

Examples
-----
:::

# set the locations of the radial gridlines
lines, labels = rgrids( (0.25, 0.5, 1.0) )

# set the locations and labels of the radial gridlines
lines, labels = rgrids( (0.25, 0.5, 1.0), ('Tom', 'Dick', 'Harry') )
"""
ax = gca()
if not isinstance(ax, PolarAxes):
    raise RuntimeError('rgrids only defined for polar axes')
if not args and not kwargs:
    lines = ax.yaxis.get_gridlines()
    labels = ax.yaxis.get_ticklabels()
else:
    lines, labels = ax.set_rgrids(*args, **kwargs)
return lines, labels

def thetagrids(*args, **kwargs):
    """
    Get or set the theta gridlines on the current polar plot.

    Call signatures::

        lines, labels = thetagrids()
        lines, labels = thetagrids(angles, labels=None, fmt=None, **kwargs)

    When called with no arguments, `thetagrids` simply returns the tuple
    (*lines*, *labels*). When called with arguments, the labels will
    appear at the specified angles.

Parameters
```

```
-----  
angles : tuple with floats, degrees  
    The angles of the theta gridlines.  
  
labels : tuple with strings or None  
    The labels to use at each radial gridline. The  
    `~.projections.polar.ThetaFormatter` will be used if None.  
  
fmt : str or None  
    Format string used in `~matplotlib.ticker.FormatStrFormatter`.  
    For example '%f'. Note that the angle in radians will be used.
```

Returns

```
-----  
lines : list of `~.lines.Line2D`  
    The theta gridlines.  
  
labels : list of `~.text.Text`  
    The tick labels.
```

Other Parameters

```
-----  
**kwargs  
    *kwargs* are optional `~.Text` properties for the labels.
```

See Also

```
-----  
.pyplot.rgrids  
.projections.polar.PolarAxes.set_thetagrids  
.Axis.get_gridlines  
.Axis.get_ticklabels
```

Examples

```
-----  
:::  
  
# set the locations of the angular gridlines  
lines, labels = thetagrids(range(45, 360, 90))  
  
# set the locations and labels of the angular gridlines  
lines, labels = thetagrids(range(45, 360, 90), ('NE', 'NW', 'SW', 'SE'))  
"""  
ax = gca()  
if not isinstance(ax, PolarAxes):  
    raise RuntimeError('thetagrids only defined for polar axes')  
if not args and not kwargs:  
    lines = ax.xaxis.get_ticklines()  
    labels = ax.xaxis.get_ticklabels()  
else:  
    lines, labels = ax.set_thetagrids(*args, **kwargs)  
return lines, labels
```

```
## Plotting Info ##
```

```
def plotting():  
    pass
```

```
def get_plot_commands():  
    """  
    Get a sorted list of all of the plotting commands.
```

```

"""
# This works by searching for all functions in this module and removing
# a few hard-coded exclusions, as well as all of the colormap-setting
# functions, and anything marked as private with a preceding underscore.
exclude = {'colormaps', 'colors', 'connect', 'disconnect',
            'get_plot_commands', 'get_current_fig_manager', 'ginput',
            'plotting', 'waitforbuttonpress'}
exclude |= set(colormaps())
this_module = inspect.getmodule(get_plot_commands)
return sorted(
    name for name, obj in globals().items()
    if not name.startswith('_') and name not in exclude
        and inspect.isfunction(obj)
        and inspect.getmodule(obj) is this_module)

```

def colormaps():

```

"""
Matplotlib provides a number of colormaps, and others can be added using
:func:`~matplotlib.cm.register_cmap`. This function documents the built-in
colormaps, and will also return a list of all registered colormaps if
called.

```

You can set the colormap for an image, pcolor, scatter, etc, using a keyword argument::

```
imshow(X, cmap=cm.hot)
```

or using the :func:`set_cmap` function::

```
imshow(X)
pyplot.set_cmap('hot')
pyplot.set_cmap('jet')
```

In interactive mode, :func:`set_cmap` will update the colormap post-hoc, allowing you to see which one works best for your data.

All built-in colormaps can be reversed by appending ``_r``: For instance, ``gray_r`` is the reverse of ``gray``.

There are several common color schemes used in visualization:

Sequential schemes

for unipolar data that progresses from low to high

Diverging schemes

for bipolar data that emphasizes positive or negative deviations from a central value

Cyclic schemes

for plotting values that wrap around at the endpoints, such as phase angle, wind direction, or time of day

Qualitative schemes

for nominal data that has no inherent ordering, where color is used only to distinguish categories

Matplotlib ships with 4 perceptually uniform color maps which are the recommended color maps for sequential data:

| Colormap | Description |
|----------|---|
| inferno | perceptually uniform shades of black-red-yellow |
| magma | perceptually uniform shades of black-red-white |
| plasma | perceptually uniform shades of blue-red-yellow |

```
viridis      perceptually uniform shades of blue-green-yellow
=====      =====
```

The following colormaps are based on the `ColorBrewer
<<https://colorbrewer2.org>>`_ color specifications and designs developed by Cynthia Brewer:

ColorBrewer Diverging (luminance is highest at the midpoint, and decreases towards differently-colored endpoints):

```
=====      =====
Colormap  Description
=====      =====
BrBG      brown, white, blue-green
PiYG      pink, white, yellow-green
PRGn      purple, white, green
PuOr      orange, white, purple
RdBu      red, white, blue
RdGy      red, white, gray
RdYlBu    red, yellow, blue
RdYlGn    red, yellow, green
Spectral   red, orange, yellow, green, blue
=====      =====
```

ColorBrewer Sequential (luminance decreases monotonically):

```
=====      =====
Colormap  Description
=====      =====
Blues     white to dark blue
BuGn      white, light blue, dark green
BuPu      white, light blue, dark purple
GnBu      white, light green, dark blue
Greens    white to dark green
Greys     white to black (not linear)
Oranges   white, orange, dark brown
OrRd      white, orange, dark red
PuBu      white, light purple, dark blue
PuBuGn   white, light purple, dark green
PuRd      white, light purple, dark red
Purples   white to dark purple
RdPu      white, pink, dark purple
Reds      white to dark red
YlGn      light yellow, dark green
YlGnBu   light yellow, light green, dark blue
YlOrBr   light yellow, orange, dark brown
YlOrRd   light yellow, orange, dark red
=====      =====
```

ColorBrewer Qualitative:

(For plotting nominal data, `ListedColormap` is used, not `LinearSegmentedColormap`. Different sets of colors are recommended for different numbers of categories.)

- * Accent
- * Dark2
- * Paired
- * Pastel1
- * Pastel2
- * Set1
- * Set2
- * Set3

A set of colormaps derived from those of the same name provided with Matlab are also included:

| Colormap | Description |
|----------|---|
| autumn | sequential linearly-increasing shades of red-orange-yellow |
| bone | sequential increasing black-white color map with a tinge of blue, to emulate X-ray film |
| cool | linearly-decreasing shades of cyan-magenta |
| copper | sequential increasing shades of black-copper |
| flag | repetitive red-white-blue-black pattern (not cyclic at endpoints) |
| gray | sequential linearly-increasing black-to-white grayscale |
| hot | sequential black-red-yellow-white, to emulate blackbody radiation from an object at increasing temperatures |
| jet | a spectral map with dark endpoints, blue-cyan-yellow-red; based on a fluid-jet simulation by NCSA [#] |
| pink | sequential increasing pastel black-pink-white, meant for sepia tone colorization of photographs |
| prism | repetitive red-yellow-green-blue-purple-...-green pattern (not cyclic at endpoints) |
| spring | linearly-increasing shades of magenta-yellow |
| summer | sequential linearly-increasing shades of green-yellow |
| winter | linearly-increasing shades of blue-green |

A set of palettes from the `Yorick scientific visualisation package <<https://dmunro.github.io/yorick-doc/>>`, an evolution of the GIST package, both by David H. Munro are included:

| Colormap | Description |
|--------------|---|
| gist_earth | mapmaker's colors from dark blue deep ocean to green lowlands to brown highlands to white mountains |
| gist_heat | sequential increasing black-red-orange-white, to emulate blackbody radiation from an iron bar as it grows hotter |
| gist_ncar | pseudo-spectral black-blue-green-yellow-red-purple-white colormap from National Center for Atmospheric Research [#] |
| gist_rainbow | runs through the colors in spectral order from red to violet at full saturation (like *hsv* but not cyclic) |
| gist_stern | "Stern special" color table from Interactive Data Language software |

A set of cyclic color maps:

| Colormap | Description |
|------------------|---|
| hsv | red-yellow-green-cyan-blue-magenta-red, formed by changing the hue component in the HSV color space |
| twilight | perceptually uniform shades of white-blue-black-red-white |
| twilight_shifted | perceptually uniform shades of black-blue-white-red-black |

Other miscellaneous schemes:

```

=====
Colormap      Description
=====
afmhot        sequential black-orange-yellow-white blackbody
               spectrum, commonly used in atomic force microscopy
brg           blue-red-green
bwr           diverging blue-white-red
coolwarm      diverging blue-gray-red, meant to avoid issues with 3D
               shading, color blindness, and ordering of colors [#]_
CMRmap        "Default colormaps on color images often reproduce to
               confusing grayscale images. The proposed colormap
               maintains an aesthetically pleasing color image that
               automatically reproduces to a monotonic grayscale with
               discrete, quantifiable saturation levels." [#]_
cubehelix     Unlike most other color schemes cubehelix was designed
               by D.A. Green to be monotonically increasing in terms
               of perceived brightness. Also, when printed on a black
               and white postscript printer, the scheme results in a
               greyscale with monotonically increasing brightness.
               This color scheme is named cubehelix because the (r, g, b)
               values produced can be visualised as a squashed helix
               around the diagonal in the (r, g, b) color cube.
gnuplot       gnuplot's traditional pm3d scheme
               (black-blue-red-yellow)
gnuplot2      sequential color printable as gray
               (black-blue-violet-yellow-white)
ocean          green-blue-white
rainbow        spectral purple-blue-green-yellow-orange-red colormap
               with diverging luminance
seismic        diverging blue-white-red
nipy_spectral  black-purple-blue-green-yellow-red-white spectrum,
               originally from the Neuroimaging in Python project
terrain        mapmaker's colors, blue-green-yellow-brown-white,
               originally from IGOR Pro
=====
```

The following colormaps are redundant and may be removed in future versions. It's recommended to use the names in the descriptions instead, which produce identical output:

```

=====
Colormap      Description
=====
gist_gray     identical to *gray*
gist_yarg     identical to *gray_r*
binary        identical to *gray_r*
=====
```

.. rubric:: Footnotes

.. [#] Rainbow colormaps, ``jet`` in particular, are considered a poor choice for scientific visualization by many researchers: `Rainbow Color Map (Still) Considered Harmful <<https://ieeexplore.ieee.org/document/4118486/?arnumber=4118486>>`_

.. [#] Resembles "BkBlAqGrYeOrReViWh200" from NCAR Command Language. See `Color Table Gallery <https://www.ncl.ucar.edu/Document/Graphics/color_table_gallery.shtml>`_

.. [#] See `Diverging Color Maps for Scientific Visualization <<http://www.kennethmoreland.com/color-maps/>>`_ by Kenneth Moreland.

```

... [#] See `A Color Map for Effective Black-and-White Rendering of
Color-Scale Images
<https://www.mathworks.com/matlabcentral/fileexchange/2662-cmrmap-m>`_
by Carey Rappaport
"""
return sorted(cm.cmap_d)

def _setup_pyplot_info_docstrings():
    """
    Generate the plotting docstring.

    These must be done after the entire module is imported, so it is
    called from the end of this module, which is generated by
    boilerplate.py.
    """
    commands = get_plot_commands()

    first_sentence = re.compile(r"(?:\s*).+?\.(?:\s+|\$)", flags=re.DOTALL)

    # Collect the first sentence of the docstring for all of the
    # plotting commands.
    rows = []
    max_name = len("Function")
    max_summary = len("Description")
    for name in commands:
        doc = globals()[name].__doc__
        summary = ''
        if doc is not None:
            match = first_sentence.match(doc)
            if match is not None:
                summary = inspect.cleandoc(match.group(0)).replace('\n', ' ')
        name = '`%s`' % name
        rows.append([name, summary])
        max_name = max(max_name, len(name))
        max_summary = max(max_summary, len(summary))

    separator = '=' * max_name + ' ' + '=' * max_summary
    lines = [
        separator,
        '{:{} {}:{}}'.format('Function', max_name, 'Description', max_summary),
        separator,
    ] + [
        '{:{} {}:{}}'.format(name, max_name, summary, max_summary)
        for name, summary in rows
    ] + [
        separator,
    ]
    plotting.__doc__ = '\n'.join(lines)

## Plotting part 1: manually generated functions and wrappers ##

def colorbar(mappable=None, cax=None, ax=None, **kw):
    if mappable is None:
        mappable = gci()
    if mappable is None:
        raise RuntimeError('No mappable was found to use for colorbar '
                           'creation. First define a mappable such as '
                           'an image (with imshow) or a contour set ('
                           'with contourf).')
    if ax is None:

```

```
    ax = gca()
    ret = gcf().colorbar(mappable, cax=cax, ax=ax, **kw)
    return ret
colorbar.__doc__ = matplotlib.colorbar.colorbar_doc

def clim(vmin=None, vmax=None):
    """
    Set the color limits of the current image.

    If either *vmin* or *vmax* is None, the image min/max respectively
    will be used for color scaling.

    If you want to set the clim of multiple images, use
    `~.ScalarMappable.set_clim` on every image, for example::

        for im in gca().get_images():
            im.set_clim(0, 0.5)

    """
    im = gci()
    if im is None:
        raise RuntimeError('You must first define an image, e.g., with imshow')

    im.set_clim(vmin, vmax)

def set_cmap(cmap):
    """
    Set the default colormap, and applies it to the current image if any.

    Parameters
    -----
    cmap : `~matplotlib.colors.Colormap` or str
        A colormap instance or the name of a registered colormap.

    See Also
    -----
    colormaps
    matplotlib.cm.register_cmap
    matplotlib.cm.get_cmap
    """
    cmap = cm.get_cmap(cmap)

    rc('image', cmap=cmap.name)
    im = gci()

    if im is not None:
        im.set_cmap(cmap)

 @_copy_docstring_and_deprecators(matplotlib.image.imread)
def imread(fname, format=None):
    return matplotlib.image.imread(fname, format)

 @_copy_docstring_and_deprecators(matplotlib.image.imsave)
def imsave(fname, arr, **kwargs):
    return matplotlib.image.imsave(fname, arr, **kwargs)

def matshow(A, fignum=None, **kwargs):
    """
```

```
Display an array as a matrix in a new figure window.
```

The origin is set at the upper left hand corner and rows (first dimension of the array) are displayed horizontally. The aspect ratio of the figure window is that of the array, unless this would make an excessively short or narrow figure.

Tick labels for the xaxis are placed on top.

Parameters

A : array-like(M, N)

The matrix to be displayed.

fignum : None or int or False

If *None*, create a new figure window with automatic numbering.

If a nonzero integer, draw into the figure with the given number
(create it if it does not exist).

If 0, use the current axes (or create one if it does not exist).

.. note::

Because of how `Axes.matshow` tries to set the figure aspect ratio to be the one of the array, strange things may happen if you reuse an existing figure.

Returns

`~matplotlib.image.AxesImage`

Other Parameters

**kwargs : `~matplotlib.axes.Axes.imshow` arguments

"""

```
A = np.asarray(A)
if fignum == 0:
    ax = gca()
else:
    # Extract actual aspect ratio of array and make appropriately sized
    # figure.
    fig = figure(fignum, figsize=figaspect(A))
    ax = fig.add_axes([0.15, 0.09, 0.775, 0.775])
im = ax.matshow(A, **kwargs)
sci(im)
return im
```

```
def polar(*args, **kwargs):
```

"""

Make a polar plot.

call signature::

```
polar(theta, r, **kwargs)
```

Multiple *theta*, *r* arguments are supported, with format strings, as in
`plot`.

"""

```
# If an axis already exists, check if it has a polar projection
if gcf().get_axes():
```

```

    if not isinstance(gca(), PolarAxes):
        cbook._warn_external('Trying to create polar plot on an axis '
                             'that does not have a polar projection.')
    ax = gca(polar=True)
    ret = ax.plot(*args, **kwargs)
    return ret

# If rcParams['backend_fallback'] is true, and an interactive backend is
# requested, ignore rcParams['backend'] and force selection of a backend that
# is compatible with the current running interactive framework.
if (rcParams["backend_fallback"]
    and dict.__getitem__(rcParams, "backend") in (
        set(_interactive_bk) - {'WebAgg', 'nbAgg'})
    and cbook._get_running_interactive_framework()):
    dict.__setitem__(rcParams, "backend", rcsetup._auto_backend_sentinel)
# Set up the backend.
switch_backend(rcParams["backend"])

# Just to be safe. Interactive mode can be turned on without
# calling `plt.ion()` so register it again here.
# This is safe because multiple calls to `install_repl_displayhook`
# are no-ops and the registered function respect `mpl.is_interactive()`
# to determine if they should trigger a draw.
install_repl_displayhook()

#####
# REMAINING CONTENT GENERATED BY boilerplate.py #####
# Autogenerated by boilerplate.py. Do not edit as changes will be lost.
 @_copy_docstring_and_deprecators(Figure.figimage)
def figimage(
    X, xo=0, yo=0, alpha=None, norm=None, cmap=None, vmin=None,
    vmax=None, origin=None, resize=False, **kwargs):
    return gcf().figimage(
        X, xo=xo, yo=yo, alpha=alpha, norm=norm, cmap=cmap, vmin=vmin,
        vmax=vmax, origin=origin, resize=resize, **kwargs)

# Autogenerated by boilerplate.py. Do not edit as changes will be lost.
 @_copy_docstring_and_deprecators(Figure.text)
def figtext(x, y, s, fontdict=None, **kwargs):
    return gcf().text(x, y, s, fontdict=fontdict, **kwargs)

# Autogenerated by boilerplate.py. Do not edit as changes will be lost.
 @_copy_docstring_and_deprecators(Figure.gca)
def gca(**kwargs):
    return gcf().gca(**kwargs)

# Autogenerated by boilerplate.py. Do not edit as changes will be lost.
 @_copy_docstring_and_deprecators(Figure._gci)
def gci():
    return gcf()._gci()

# Autogenerated by boilerplate.py. Do not edit as changes will be lost.
 @_copy_docstring_and_deprecators(Figure.ginput)
def ginput(
    n=1, timeout=30, show_clicks=True,
    mouse_add=MouseButton.LEFT, mouse_pop=MouseButton.RIGHT,

```

```
    mouse_stop=MouseButton.MIDDLE):
    return gcf().ginput(
        n=n, timeout=timeout, show_clicks=show_clicks,
        mouse_add=mouse_add, mouse_pop=mouse_pop,
        mouse_stop=mouse_stop)

# Autogenerated by boilerplate.py. Do not edit as changes will be lost.
 @_copy_docstring_and_deprecators(Figure.subplots_adjust)
def subplots_adjust(
    left=None, bottom=None, right=None, top=None, wspace=None,
    hspace=None):
    return gcf().subplots_adjust(
        left=left, bottom=bottom, right=right, top=top, wspace=wspace,
        hspace=hspace)

# Autogenerated by boilerplate.py. Do not edit as changes will be lost.
 @_copy_docstring_and_deprecators(Figure.suptitle)
def suptitle(t, **kwargs):
    return gcf().suptitle(t, **kwargs)

# Autogenerated by boilerplate.py. Do not edit as changes will be lost.
 @_copy_docstring_and_deprecators(Figure.waitforbuttonpress)
def waitforbuttonpress(timeout=-1):
    return gcf().waitforbuttonpress(timeout=timeout)

# Autogenerated by boilerplate.py. Do not edit as changes will be lost.
 @_copy_docstring_and_deprecators(Axes.acorr)
def acorr(x, *, data=None, **kwargs):
    return gca().acorr(
        x, **({"data": data} if data is not None else {}), **kwargs)

# Autogenerated by boilerplate.py. Do not edit as changes will be lost.
 @_copy_docstring_and_deprecators(Axes.angle_spectrum)
def angle_spectrum(
    x, Fs=None, Fc=None, window=None, pad_to=None, sides=None, *,
    data=None, **kwargs):
    return gca().angle_spectrum(
        x, Fs=Fs, Fc=Fc, window=window, pad_to=pad_to, sides=sides,
        **({"data": data} if data is not None else {}), **kwargs)

# Autogenerated by boilerplate.py. Do not edit as changes will be lost.
 @_copy_docstring_and_deprecators(Axes.annotate)
def annotate(text, xy, *args, **kwargs):
    return gca().annotate(text, xy, *args, **kwargs)

# Autogenerated by boilerplate.py. Do not edit as changes will be lost.
 @_copy_docstring_and_deprecators(Axes.arrow)
def arrow(x, y, dx, dy, **kwargs):
    return gca().arrow(x, y, dx, dy, **kwargs)

# Autogenerated by boilerplate.py. Do not edit as changes will be lost.
 @_copy_docstring_and_deprecators(Axes.autoscale)
def autoscale(enable=True, axis='both', tight=None):
    return gca().autoscale(enable=enable, axis=axis, tight=tight)
```

```
# Autogenerated by boilerplate.py.  Do not edit as changes will be lost.
 @_copy_docstring_and_deprecators(Axes.axhline)
def axhline(y=0, xmin=0, xmax=1, **kwargs):
    return gca().axhline(y=y, xmin=xmin, xmax=xmax, **kwargs)

# Autogenerated by boilerplate.py.  Do not edit as changes will be lost.
 @_copy_docstring_and_deprecators(Axes.axhspan)
def axhspan(ymin, ymax, xmin=0, xmax=1, **kwargs):
    return gca().axhspan(ymin, ymax, xmin=xmin, xmax=xmax, **kwargs)

# Autogenerated by boilerplate.py.  Do not edit as changes will be lost.
 @_copy_docstring_and_deprecators(Axes.axis)
def axis(*args, emit=True, **kwargs):
    return gca().axis(*args, emit=emit, **kwargs)

# Autogenerated by boilerplate.py.  Do not edit as changes will be lost.
 @_copy_docstring_and_deprecators(Axes.axline)
def axline(xy1, xy2=None, *, slope=None, **kwargs):
    return gca().axline(xy1, xy2=xy2, slope=slope, **kwargs)

# Autogenerated by boilerplate.py.  Do not edit as changes will be lost.
 @_copy_docstring_and_deprecators(Axes.axvline)
def axvline(x=0, ymin=0, ymax=1, **kwargs):
    return gca().axvline(x=x, ymin=ymin, ymax=ymax, **kwargs)

# Autogenerated by boilerplate.py.  Do not edit as changes will be lost.
 @_copy_docstring_and_deprecators(Axes.axvspan)
def axvspan(xmin, xmax, ymin=0, ymax=1, **kwargs):
    return gca().axvspan(xmin, xmax, ymin=ymin, ymax=ymax, **kwargs)

# Autogenerated by boilerplate.py.  Do not edit as changes will be lost.
 @_copy_docstring_and_deprecators(Axes.bar)
def bar(
    x, height, width=0.8, bottom=None, *, align='center',
    data=None, **kwargs):
    return gca().bar(
        x, height, width=width, bottom=bottom, align=align,
        **({"data": data} if data is not None else {}), **kwargs)

# Autogenerated by boilerplate.py.  Do not edit as changes will be lost.
 @_copy_docstring_and_deprecators(Axes.barbs)
def barbs(*args, data=None, **kw):
    return gca().barbs(
        *args, **({"data": data} if data is not None else {}), **kw)

# Autogenerated by boilerplate.py.  Do not edit as changes will be lost.
 @_copy_docstring_and_deprecators(Axes.bart)
def bart(y, width, height=0.8, left=None, *, align='center', **kwargs):
    return gca().bart(
        y, width, height=height, left=left, align=align, **kwargs)

# Autogenerated by boilerplate.py.  Do not edit as changes will be lost.
 @_copy_docstring_and_deprecators(Axes.boxplot)
```

```

def boxplot(
    x, notch=None, sym=None, vert=None, whis=None,
    positions=None, widths=None, patch_artist=None,
    bootstrap=None, usermedians=None, conf_intervals=None,
    meanline=None, showmeans=None, showcaps=None, showbox=None,
    showfliers=None, boxprops=None, labels=None, flierprops=None,
    medianprops=None, meanprops=None, capprops=None,
    whiskerprops=None, manage_ticks=True, autorange=False,
    zorder=None, *, data=None):
    return gca().boxplot(
        x, notch=notch, sym=sym, vert=vert, whis=whis,
        positions=positions, widths=widths, patch_artist=patch_artist,
        bootstrap=bootstrap, usermedians=usermedians,
        conf_intervals=conf_intervals, meanline=meanline,
        showmeans=showmeans, showcaps=showcaps, showbox=showbox,
        showfliers=showfliers, boxprops=boxprops, labels=labels,
        flierprops=flierprops, medianprops=medianprops,
        meanprops=meanprops, capprops=capprops,
        whiskerprops=whiskerprops, manage_ticks=manage_ticks,
        autorange=autorange, zorder=zorder,
        **({"data": data} if data is not None else {}))

# Autogenerated by boilerplate.py. Do not edit as changes will be lost.
 @_copy_docstring_and_deprecators(Axes.broken_barh)
def broken_barh(xranges, yrange, *, data=None, **kwargs):
    return gca().broken_barh(
        xranges, yrange,
        **({"data": data} if data is not None else {}), **kwargs)

# Autogenerated by boilerplate.py. Do not edit as changes will be lost.
 @_copy_docstring_and_deprecators(Axes.cla)
def cla():
    return gca().cla()

# Autogenerated by boilerplate.py. Do not edit as changes will be lost.
 @_copy_docstring_and_deprecators(Axes.clabel)
def clabel(CS, levels=None, **kwargs):
    return gca().clabel(CS, levels=levels, **kwargs)

# Autogenerated by boilerplate.py. Do not edit as changes will be lost.
 @_copy_docstring_and_deprecators(Axes.cohere)
def cohere(
    x, y, NFFT=256, Fs=2, Fc=0, detrend=mlab.detrend_none,
    window=mlab.window_hanning, nooverlap=0, pad_to=None,
    sides='default', scale_by_freq=None, *, data=None, **kwargs):
    return gca().cohere(
        x, y, NFFT=NFFT, Fs=Fs, Fc=Fc, detrend=detrend, window=window,
        nooverlap=nooverlap, pad_to=pad_to, sides=sides,
        scale_by_freq=scale_by_freq,
        **({"data": data} if data is not None else {}), **kwargs)

# Autogenerated by boilerplate.py. Do not edit as changes will be lost.
 @_copy_docstring_and_deprecators(Axes.contour)
def contour(*args, data=None, **kwargs):
    __ret = gca().contour(
        *args, **({"data": data} if data is not None else {}),
        **kwargs)
    if __ret.A is not None: sci(__ret) # noqa

```

```

return __ret

# Autogenerated by boilerplate.py. Do not edit as changes will be lost.
 @_copy_docstring_and_deprecators(Axes.contourf)
def contourf(*args, data=None, **kwargs):
    __ret = gca().contourf(
        *args, **({{"data": data} if data is not None else {}}),
        **kwargs)
    if __ret._A is not None: sci(__ret) # noqa
    return __ret

# Autogenerated by boilerplate.py. Do not edit as changes will be lost.
 @_copy_docstring_and_deprecators(Axes.csd)
def csd(
    x, y, NFFT=None, Fs=None, Fc=None, detrend=None, window=None,
    nooverlap=None, pad_to=None, sides=None, scale_by_freq=None,
    return_line=None, *, data=None, **kwargs):
    return gca().csd(
        x, y, NFFT=NFFT, Fs=Fs, Fc=Fc, detrend=detrend, window=window,
        nooverlap=nooverlap, pad_to=pad_to, sides=sides,
        scale_by_freq=scale_by_freq, return_line=return_line,
        **({{"data": data} if data is not None else {}}), **kwargs)

# Autogenerated by boilerplate.py. Do not edit as changes will be lost.
 @_copy_docstring_and_deprecators(Axes.errorbar)
def errorbar(
    x, y, yerr=None, xerr=None, fmt='', ecolor=None,
    elinewidth=None, capsizes=None, barsabove=False, lolims=False,
    uplims=False, xlolims=False, xuplims=False, errorevery=1,
    capthick=None, *, data=None, **kwargs):
    return gca().errorbar(
        x, y, yerr=yerr, xerr=xerr, fmt=fmt, ecolor=ecolor,
        elinewidth=elinewidth, capsizes=capsizes, barsabove=barsabove,
        lolims=lolims, uplims=uplims, xlolims=xlolims,
        xuplims=xuplims, errorevery=errorevery, capthick=capthick,
        **({{"data": data} if data is not None else {}}), **kwargs)

# Autogenerated by boilerplate.py. Do not edit as changes will be lost.
 @_copy_docstring_and_deprecators(Axes.eventplot)
def eventplot(
    positions, orientation='horizontal', lineoffsets=1,
    linelengths=1, linewidths=None, colors=None,
    linestyles='solid', *, data=None, **kwargs):
    return gca().eventplot(
        positions, orientation=orientation, lineoffsets=lineoffsets,
        linelengths=linelengths, linewidths=linewidths, colors=colors,
        linestyles=linestyles,
        **({{"data": data} if data is not None else {}}), **kwargs)

# Autogenerated by boilerplate.py. Do not edit as changes will be lost.
 @_copy_docstring_and_deprecators(Axes.fill)
def fill(*args, data=None, **kwargs):
    return gca().fill(
        *args, **({{"data": data} if data is not None else {}}),
        **kwargs)

# Autogenerated by boilerplate.py. Do not edit as changes will be lost.

```

```

 @_copy_docstring_and_deprecators(Axes.fill_between)
def fill_between(
    x, y1, y2=0, where=None, interpolate=False, step=None, *,
    data=None, **kwargs):
    return gca().fill_between(
        x, y1, y2=y2, where=where, interpolate=interpolate, step=step,
        **({"data": data} if data is not None else {}), **kwargs)

# Autogenerated by boilerplate.py. Do not edit as changes will be lost.
 @_copy_docstring_and_deprecators(Axes.fill_betweenx)
def fill_betweenx(
    y, x1, x2=0, where=None, step=None, interpolate=False, *,
    data=None, **kwargs):
    return gca().fill_betweenx(
        y, x1, x2=x2, where=where, step=step, interpolate=interpolate,
        **({"data": data} if data is not None else {}), **kwargs)

# Autogenerated by boilerplate.py. Do not edit as changes will be lost.
 @_copy_docstring_and_deprecators(Axes.grid)
def grid(b=None, which='major', axis='both', **kwargs):
    return gca().grid(b=b, which=which, axis=axis, **kwargs)

# Autogenerated by boilerplate.py. Do not edit as changes will be lost.
 @_copy_docstring_and_deprecators(Axes.hexbin)
def hexbin(
    x, y, C=None, gridsize=100, bins=None, xscale='linear',
    yscale='linear', extent=None, cmap=None, norm=None, vmin=None,
    vmax=None, alpha=None, linewidths=None, edgecolors='face',
    reduce_C_function=np.mean, mincnt=None, marginals=False, *,
    data=None, **kwargs):
    __ret = gca().hexbin(
        x, y, C=C, gridsize=gridsize, bins=bins, xscale=xscale,
        yscale=yscale, extent=extent, cmap=cmap, norm=norm, vmin=vmin,
        vmax=vmax, alpha=alpha, linewidths=linewidths,
        edgecolors=edgecolors, reduce_C_function=reduce_C_function,
        mincnt=mincnt, marginals=marginals,
        **({"data": data} if data is not None else {}), **kwargs)
    sci(__ret)
    return __ret

# Autogenerated by boilerplate.py. Do not edit as changes will be lost.
 @_copy_docstring_and_deprecators(Axes.hist)
def hist(
    x, bins=None, range=None, density=False, weights=None,
    cumulative=False, bottom=None, histtype='bar', align='mid',
    orientation='vertical', rwidth=None, log=False, color=None,
    label=None, stacked=False, *, data=None, **kwargs):
    return gca().hist(
        x, bins=bins, range=range, density=density, weights=weights,
        cumulative=cumulative, bottom=bottom, histtype=histtype,
        align=align, orientation=orientation, rwidth=rwidth, log=log,
        color=color, label=label, stacked=stacked,
        **({"data": data} if data is not None else {}), **kwargs)

# Autogenerated by boilerplate.py. Do not edit as changes will be lost.
 @_copy_docstring_and_deprecators(Axes.hist2d)
def hist2d(
    x, y, bins=10, range=None, density=False, weights=None,

```

```

    cmin=None, cmax=None, *, data=None, **kwargs):
    __ret = gca().hist2d(
        x, y, bins=bins, range=range, density=density,
        weights=weights, cmin=cmin, cmax=cmax,
        **({ "data": data} if data is not None else {}), **kwargs)
    sci(__ret[-1])
    return __ret

# Autogenerated by boilerplate.py. Do not edit as changes will be lost.
 @_copy_docstring_and_deprecators(Axes.hlines)
def hlines(
    y, xmin, xmax, colors='k', linestyles='solid', label='', *,
    data=None, **kwargs):
    return gca().hlines(
        y, xmin, xmax, colors=colors, linestyles=linestyles,
        label=label, **({ "data": data} if data is not None else {}),
        **kwargs)

# Autogenerated by boilerplate.py. Do not edit as changes will be lost.
 @_copy_docstring_and_deprecators(Axes.imshow)
def imshow(
    X, cmap=None, norm=None, aspect=None, interpolation=None,
    alpha=None, vmin=None, vmax=None, origin=None, extent=None, *,
    filternorm=True, filterrad=4.0, resample=None, url=None,
    data=None, **kwargs):
    __ret = gca().imshow(
        X, cmap=cmap, norm=norm, aspect=aspect,
        interpolation=interpolation, alpha=alpha, vmin=vmin,
        vmax=vmax, origin=origin, extent=extent,
        filternorm=filternorm, filterrad=filterrad, resample=resample,
        url=url, **({ "data": data} if data is not None else {}),
        **kwargs)
    sci(__ret)
    return __ret

# Autogenerated by boilerplate.py. Do not edit as changes will be lost.
 @_copy_docstring_and_deprecators(Axes.legend)
def legend(*args, **kwargs):
    return gca().legend(*args, **kwargs)

# Autogenerated by boilerplate.py. Do not edit as changes will be lost.
 @_copy_docstring_and_deprecators(Axes.locator_params)
def locator_params(axis='both', tight=None, **kwargs):
    return gca().locator_params(axis=axis, tight=tight, **kwargs)

# Autogenerated by boilerplate.py. Do not edit as changes will be lost.
 @_copy_docstring_and_deprecators(Axes.loglog)
def loglog(*args, **kwargs):
    return gca().loglog(*args, **kwargs)

# Autogenerated by boilerplate.py. Do not edit as changes will be lost.
 @_copy_docstring_and_deprecators(Axes.magnitude_spectrum)
def magnitude_spectrum(
    x, Fs=None, Fc=None, window=None, pad_to=None, sides=None,
    scale=None, *, data=None, **kwargs):
    return gca().magnitude_spectrum(
        x, Fs=Fs, Fc=Fc, window=window, pad_to=pad_to, sides=sides,

```

```
scale=scale, **({"data": data} if data is not None else {}),  
**kwargs)  
  
# Autogenerated by boilerplate.py. Do not edit as changes will be lost.  
 @_copy_docstring_and_deprecators(Axes.margins)  
def margins(*margins, x=None, y=None, tight=True):  
    return gca().margins(*margins, x=x, y=y, tight=tight)  
  
# Autogenerated by boilerplate.py. Do not edit as changes will be lost.  
 @_copy_docstring_and_deprecators(Axes.minorticks_off)  
def minorticks_off():  
    return gca().minorticks_off()  
  
# Autogenerated by boilerplate.py. Do not edit as changes will be lost.  
 @_copy_docstring_and_deprecators(Axes.minorticks_on)  
def minorticks_on():  
    return gca().minorticks_on()  
  
# Autogenerated by boilerplate.py. Do not edit as changes will be lost.  
 @_copy_docstring_and_deprecators(Axes.pcolor)  
def pcolor(  
    *args, shading=None, alpha=None, norm=None, cmap=None,  
    vmin=None, vmax=None, data=None, **kwargs):  
    __ret = gca().pcolor(  
        *args, shading=shading, alpha=alpha, norm=norm, cmap=cmap,  
        vmin=vmin, vmax=vmax,  
        **({"data": data} if data is not None else {}), **kwargs)  
    scи(__ret)  
    return __ret  
  
# Autogenerated by boilerplate.py. Do not edit as changes will be lost.  
 @_copy_docstring_and_deprecators(Axes.pcolormesh)  
def pcolormesh(  
    *args, alpha=None, norm=None, cmap=None, vmin=None,  
    vmax=None, shading=None, antialiased=False, data=None,  
    **kwargs):  
    __ret = gca().pcolormesh(  
        *args, alpha=alpha, norm=norm, cmap=cmap, vmin=vmin,  
        vmax=vmax, shading=shading, antialiased=antialiased,  
        **({"data": data} if data is not None else {}), **kwargs)  
    scи(__ret)  
    return __ret  
  
# Autogenerated by boilerplate.py. Do not edit as changes will be lost.  
 @_copy_docstring_and_deprecators(Axes.phase_spectrum)  
def phase_spectrum(  
    x, Fs=None, Fc=None, window=None, pad_to=None, sides=None, *,  
    data=None, **kwargs):  
    return gca().phase_spectrum(  
        x, Fs=Fs, Fc=Fc, window=window, pad_to=pad_to, sides=sides,  
        **({"data": data} if data is not None else {}), **kwargs)  
  
# Autogenerated by boilerplate.py. Do not edit as changes will be lost.  
 @_copy_docstring_and_deprecators(Axes.pie)  
def pie(  
    x, explode=None, labels=None, colors=None, autopct=None,
```

```

pctdistance=0.6, shadow=False, labeldistance=1.1,
startangle=0, radius=1, counterclock=True, wedgeprops=None,
textprops=None, center=(0, 0), frame=False,
rotatelabels=False, *, data=None):
    return gca().pie(
        x, explode=explode, labels=labels, colors=colors,
        autopct=autopct, pctdistance=pctdistance, shadow=shadow,
        labeldistance=labeldistance, startangle=startangle,
        radius=radius, counterclock=counterclock,
        wedgeprops=wedgeprops, textprops=textprops, center=center,
        frame=frame, rotatelabels=rotatelabels,
        **({"data": data} if data is not None else {}))

# Autogenerated by boilerplate.py. Do not edit as changes will be lost.
 @_copy_docstring_and_deprecators(Axes.plot)
def plot(*args, scalex=True, scaley=True, data=None, **kwargs):
    return gca().plot(
        *args, scalex=scalex, scaley=scaley,
        **({"data": data} if data is not None else {}), **kwargs)

# Autogenerated by boilerplate.py. Do not edit as changes will be lost.
 @_copy_docstring_and_deprecators(Axes.plot_date)
def plot_date(
    x, y, fmt='o', tz=None, xdate=True, ydate=False, *,
    data=None, **kwargs):
    return gca().plot_date(
        x, y, fmt=fmt, tz=tz, xdate=xdate, ydate=ydate,
        **({"data": data} if data is not None else {}), **kwargs)

# Autogenerated by boilerplate.py. Do not edit as changes will be lost.
 @_copy_docstring_and_deprecators(Axes.psd)
def psd(
    x, NFFT=None, Fs=None, Fc=None, detrend=None, window=None,
    nooverlap=None, pad_to=None, sides=None, scale_by_freq=None,
    return_line=None, *, data=None, **kwargs):
    return gca().psd(
        x, NFFT=NFFT, Fs=Fs, Fc=Fc, detrend=detrend, window=window,
        nooverlap=nooverlap, pad_to=pad_to, sides=sides,
        scale_by_freq=scale_by_freq, return_line=return_line,
        **({"data": data} if data is not None else {}), **kwargs)

# Autogenerated by boilerplate.py. Do not edit as changes will be lost.
 @_copy_docstring_and_deprecators(Axes.quiver)
def quiver(*args, data=None, **kw):
    __ret = gca().quiver(
        *args, **({"data": data} if data is not None else {}), **kw)
    scil(__ret)
    return __ret

# Autogenerated by boilerplate.py. Do not edit as changes will be lost.
 @_copy_docstring_and_deprecators(Axes.quiverkey)
def quiverkey(Q, X, Y, U, label, **kw):
    return gca().quiverkey(Q, X, Y, U, label, **kw)

# Autogenerated by boilerplate.py. Do not edit as changes will be lost.
 @_copy_docstring_and_deprecators(Axes.scatter)
def scatter(

```

```

        x, y, s=None, c=None, marker=None, cmap=None, norm=None,
        vmin=None, vmax=None, alpha=None, linewidths=None,
        verts=cbook.deprecation._deprecated_parameter,
        edgecolors=None, *, plotnonfinite=False, data=None, **kwargs):
    __ret = gca().scatter(
        x, y, s=s, c=c, marker=marker, cmap=cmap, norm=norm,
        vmin=vmin, vmax=vmax, alpha=alpha, linewidths=linewidths,
        verts=verts, edgecolors=edgecolors,
        plotnonfinite=plotnonfinite,
        **({"data": data} if data is not None else {}), **kwargs)
sci(__ret)
return __ret

# Autogenerated by boilerplate.py. Do not edit as changes will be lost.
 @_copy_docstring_and_deprecators(Axes.semilogx)
def semilogx(*args, **kwargs):
    return gca().semilogx(*args, **kwargs)

# Autogenerated by boilerplate.py. Do not edit as changes will be lost.
 @_copy_docstring_and_deprecators(Axes.semilogy)
def semilogy(*args, **kwargs):
    return gca().semilogy(*args, **kwargs)

# Autogenerated by boilerplate.py. Do not edit as changes will be lost.
 @_copy_docstring_and_deprecators(Axes.specgram)
def specgram(
    x, NFFT=None, Fs=None, Fc=None, detrend=None, window=None,
    nooverlap=None, cmap=None, xextent=None, pad_to=None,
    sides=None, scale_by_freq=None, mode=None, scale=None,
    vmin=None, vmax=None, *, data=None, **kwargs):
    __ret = gca().specgram(
        x, NFFT=NFFT, Fs=Fs, Fc=Fc, detrend=detrend, window=window,
        nooverlap=nooverlap, cmap=cmap, xextent=xextent, pad_to=pad_to,
        sides=sides, scale_by_freq=scale_by_freq, mode=mode,
        scale=scale, vmin=vmin, vmax=vmax,
        **({"data": data} if data is not None else {}), **kwargs)
sci(__ret[-1])
return __ret

# Autogenerated by boilerplate.py. Do not edit as changes will be lost.
 @_copy_docstring_and_deprecators(Axes.spy)
def spy(
    Z, precision=0, marker=None, markersize=None, aspect='equal',
    origin='upper', **kwargs):
    __ret = gca().spy(
        Z, precision=precision, marker=marker, markersize=markersize,
        aspect=aspect, origin=origin, **kwargs)
if isinstance(__ret, cm.ScalarMappable): sci(__ret) # noqa
return __ret

# Autogenerated by boilerplate.py. Do not edit as changes will be lost.
 @_copy_docstring_and_deprecators(Axes.stackplot)
def stackplot(
    x, *args, labels=(), colors=None, baseline='zero', data=None,
    **kwargs):
    return gca().stackplot(
        x, *args, labels=labels, colors=colors, baseline=baseline,
        **({"data": data} if data is not None else {}), **kwargs)

```

```

# Autogenerated by boilerplate.py. Do not edit as changes will be lost.
 @_copy_docstring_and_deprecators(Axes.stem)
def stem(
    *args, linefmt=None, markerfmt=None, basefmt=None, bottom=0,
    label=None, use_line_collection=True, data=None):
    return gca().stem(
        *args, linefmt=linefmt, markerfmt=markerfmt, basefmt=basefmt,
        bottom=bottom, label=label,
        use_line_collection=use_line_collection,
        **({"data": data} if data is not None else {}))

# Autogenerated by boilerplate.py. Do not edit as changes will be lost.
 @_copy_docstring_and_deprecators(Axes.step)
def step(x, y, *args, where='pre', data=None, **kwargs):
    return gca().step(
        x, y, *args, where=where,
        **({"data": data} if data is not None else {}), **kwargs)

# Autogenerated by boilerplate.py. Do not edit as changes will be lost.
 @_copy_docstring_and_deprecators(Axes.streamplot)
def streamplot(
    x, y, u, v, density=1, linewidth=None, color=None, cmap=None,
    norm=None, arrowsize=1, arrowstyle='-'|>', minlength=0.1,
    transform=None, zorder=None, start_points=None, maxlen=4.0,
    integration_direction='both', *, data=None):
    __ret = gca().streamplot(
        x, y, u, v, density=density, linewidth=linewidth, color=color,
        cmap=cmap, norm=norm, arrowsize=arrowsize,
        arrowstyle=arrowstyle, minlength=minlength,
        transform=transform, zorder=zorder, start_points=start_points,
        maxlen=maxlen,
        integration_direction=integration_direction,
        **({"data": data} if data is not None else {}))
    scilin(__ret.lines)
    return __ret

# Autogenerated by boilerplate.py. Do not edit as changes will be lost.
 @_copy_docstring_and_deprecators(Axes.table)
def table(
    cellText=None, cellColours=None, cellLoc='right',
    colWidths=None, rowLabels=None, rowColours=None,
    rowLoc='left', colLabels=None, colColours=None,
    colLoc='center', loc='bottom', bbox=None, edges='closed',
    **kwargs):
    return gca().table(
        cellText=cellText, cellColours=cellColours, cellLoc=cellLoc,
        colWidths=colWidths, rowLabels=rowLabels,
        rowColours=rowColours, rowLoc=rowLoc, colLabels=colLabels,
        colColours=colColours, colLoc=colLoc, loc=loc, bbox=bbox,
        edges=edges, **kwargs)

# Autogenerated by boilerplate.py. Do not edit as changes will be lost.
 @_copy_docstring_and_deprecators(Axes.text)
def text(x, y, s, fontdict=None, **kwargs):
    return gca().text(x, y, s, fontdict=fontdict, **kwargs)

```

```
# Autogenerated by boilerplate.py.  Do not edit as changes will be lost.
 @_copy_docstring_and_deprecators(Axes.tick_params)
def tick_params(axis='both', **kwargs):
    return gca().tick_params(axis=axis, **kwargs)

# Autogenerated by boilerplate.py.  Do not edit as changes will be lost.
 @_copy_docstring_and_deprecators(Axes.ticklabel_format)
def ticklabel_format(
    *, axis='both', style='', scilimits=None, useOffset=None,
    useLocale=None, useMathText=None):
    return gca().ticklabel_format(
        axis=axis, style=style, scilimits=scilimits,
        useOffset=useOffset, useLocale=useLocale,
        useMathText=useMathText)

# Autogenerated by boilerplate.py.  Do not edit as changes will be lost.
 @_copy_docstring_and_deprecators(Axes.tricontour)
def tricontour(*args, **kwargs):
    __ret = gca().tricontour(*args, **kwargs)
    if __ret is not None: sci(__ret) # noqa
    return __ret

# Autogenerated by boilerplate.py.  Do not edit as changes will be lost.
 @_copy_docstring_and_deprecators(Axes.tricontourf)
def tricontourf(*args, **kwargs):
    __ret = gca().tricontourf(*args, **kwargs)
    if __ret is not None: sci(__ret) # noqa
    return __ret

# Autogenerated by boilerplate.py.  Do not edit as changes will be lost.
 @_copy_docstring_and_deprecators(Axes.tripcolor)
def tripcolor(
    *args, alpha=1.0, norm=None, cmap=None, vmin=None, vmax=None,
    shading='flat', facecolors=None, **kwargs):
    __ret = gca().tripcolor(
        *args, alpha=alpha, norm=norm, cmap=cmap, vmin=vmin,
        vmax=vmax, shading=shading, facecolors=facecolors, **kwargs)
    sci(__ret)
    return __ret

# Autogenerated by boilerplate.py.  Do not edit as changes will be lost.
 @_copy_docstring_and_deprecators(Axes.triplot)
def triplot(*args, **kwargs):
    return gca().triplot(*args, **kwargs)

# Autogenerated by boilerplate.py.  Do not edit as changes will be lost.
 @_copy_docstring_and_deprecators(Axes.violinplot)
def violinplot(
    dataset, positions=None, vert=True, widths=0.5,
    showmeans=False, showextrema=True, showmedians=False,
    quantiles=None, points=100, bw_method=None, *, data=None):
    return gca().violinplot(
        dataset, positions=positions, vert=vert, widths=widths,
        showmeans=showmeans, showextrema=showextrema,
        showmedians=showmedians, quantiles=quantiles, points=points,
        bw_method=bw_method,
        **({"data": data} if data is not None else {}))
```

```
# Autogenerated by boilerplate.py.  Do not edit as changes will be lost.
 @_copy_docstring_and_deprecators(Axes.vlines)
def vlines(
    x, ymin, ymax, colors='k', linestyles='solid', label='', *,
    data=None, **kwargs):
    return gca().vlines(
        x, ymin, ymax, colors=colors, linestyles=linestyles,
        label=label, **({"data": data} if data is not None else {}),
        **kwargs)

# Autogenerated by boilerplate.py.  Do not edit as changes will be lost.
 @_copy_docstring_and_deprecators(Axes.xcorr)
def xcorr(
    x, y, normed=True, detrend=mlab.detrend_none, usevlines=True,
    maxlags=10, *, data=None, **kwargs):
    return gca().xcorr(
        x, y, normed=normed, detrend=detrend, usevlines=usevlines,
        maxlags=maxlags,
        **({"data": data} if data is not None else {}), **kwargs)

# Autogenerated by boilerplate.py.  Do not edit as changes will be lost.
 @_copy_docstring_and_deprecators(Axes._sci)
def sci(im):
    return gca()._sci(im)

# Autogenerated by boilerplate.py.  Do not edit as changes will be lost.
 @_copy_docstring_and_deprecators(Axes.set_title)
def title(label, fontdict=None, loc=None, pad=None, *, y=None, **kwargs):
    return gca().set_title(
        label, fontdict=fontdict, loc=loc, pad=pad, y=y, **kwargs)

# Autogenerated by boilerplate.py.  Do not edit as changes will be lost.
 @_copy_docstring_and_deprecators(Axes.set_xlabel)
def xlabel(xlabel, fontdict=None, labelpad=None, *, loc=None, **kwargs):
    return gca().set_xlabel(
        xlabel, fontdict=fontdict, labelpad=labelpad, loc=loc,
        **kwargs)

# Autogenerated by boilerplate.py.  Do not edit as changes will be lost.
 @_copy_docstring_and_deprecators(Axes.set_ylabel)
def ylabel(ylabel, fontdict=None, labelpad=None, *, loc=None, **kwargs):
    return gca().set_ylabel(
        ylabel, fontdict=fontdict, labelpad=labelpad, loc=loc,
        **kwargs)

# Autogenerated by boilerplate.py.  Do not edit as changes will be lost.
 @_copy_docstring_and_deprecators(Axes.set_xscale)
def xscale(value, **kwargs):
    return gca().set_xscale(value, **kwargs)

# Autogenerated by boilerplate.py.  Do not edit as changes will be lost.
 @_copy_docstring_and_deprecators(Axes.set_yscale)
def yscale(value, **kwargs):
    return gca().set_yscale(value, **kwargs)
```

```
# Autogenerated by boilerplate.py.  Do not edit as changes will be lost.
def autumn():
    """
    Set the colormap to "autumn".

    This changes the default colormap as well as the colormap of the current
    image if there is one. See ``help(colormaps)`` for more information.
    """
    set_cmap("autumn")

# Autogenerated by boilerplate.py.  Do not edit as changes will be lost.
def bone():
    """
    Set the colormap to "bone".

    This changes the default colormap as well as the colormap of the current
    image if there is one. See ``help(colormaps)`` for more information.
    """
    set_cmap("bone")

# Autogenerated by boilerplate.py.  Do not edit as changes will be lost.
def cool():
    """
    Set the colormap to "cool".

    This changes the default colormap as well as the colormap of the current
    image if there is one. See ``help(colormaps)`` for more information.
    """
    set_cmap("cool")

# Autogenerated by boilerplate.py.  Do not edit as changes will be lost.
def copper():
    """
    Set the colormap to "copper".

    This changes the default colormap as well as the colormap of the current
    image if there is one. See ``help(colormaps)`` for more information.
    """
    set_cmap("copper")

# Autogenerated by boilerplate.py.  Do not edit as changes will be lost.
def flag():
    """
    Set the colormap to "flag".

    This changes the default colormap as well as the colormap of the current
    image if there is one. See ``help(colormaps)`` for more information.
    """
    set_cmap("flag")

# Autogenerated by boilerplate.py.  Do not edit as changes will be lost.
def gray():
    """
    Set the colormap to "gray".

    This changes the default colormap as well as the colormap of the current
```

```
image if there is one. See ``help(colormaps)`` for more information.
"""
set_cmap("gray")

# Autogenerated by boilerplate.py. Do not edit as changes will be lost.
def hot():
    """
    Set the colormap to "hot".

    This changes the default colormap as well as the colormap of the current
    image if there is one. See ``help(colormaps)`` for more information.
    """
    set_cmap("hot")

# Autogenerated by boilerplate.py. Do not edit as changes will be lost.
def hsv():
    """
    Set the colormap to "hsv".

    This changes the default colormap as well as the colormap of the current
    image if there is one. See ``help(colormaps)`` for more information.
    """
    set_cmap("hsv")

# Autogenerated by boilerplate.py. Do not edit as changes will be lost.
def jet():
    """
    Set the colormap to "jet".

    This changes the default colormap as well as the colormap of the current
    image if there is one. See ``help(colormaps)`` for more information.
    """
    set_cmap("jet")

# Autogenerated by boilerplate.py. Do not edit as changes will be lost.
def pink():
    """
    Set the colormap to "pink".

    This changes the default colormap as well as the colormap of the current
    image if there is one. See ``help(colormaps)`` for more information.
    """
    set_cmap("pink")

# Autogenerated by boilerplate.py. Do not edit as changes will be lost.
def prism():
    """
    Set the colormap to "prism".

    This changes the default colormap as well as the colormap of the current
    image if there is one. See ``help(colormaps)`` for more information.
    """
    set_cmap("prism")

# Autogenerated by boilerplate.py. Do not edit as changes will be lost.
def spring():
    """
```

```
Set the colormap to "spring".

This changes the default colormap as well as the colormap of the current
image if there is one. See ``help(colormaps)`` for more information.
"""
set_cmap("spring")

# Autogenerated by boilerplate.py.  Do not edit as changes will be lost.
def summer():
    """
    Set the colormap to "summer".

    This changes the default colormap as well as the colormap of the current
    image if there is one. See ``help(colormaps)`` for more information.
    """
    set_cmap("summer")

# Autogenerated by boilerplate.py.  Do not edit as changes will be lost.
def winter():
    """
    Set the colormap to "winter".

    This changes the default colormap as well as the colormap of the current
    image if there is one. See ``help(colormaps)`` for more information.
    """
    set_cmap("winter")

# Autogenerated by boilerplate.py.  Do not edit as changes will be lost.
def magma():
    """
    Set the colormap to "magma".

    This changes the default colormap as well as the colormap of the current
    image if there is one. See ``help(colormaps)`` for more information.
    """
    set_cmap("magma")

# Autogenerated by boilerplate.py.  Do not edit as changes will be lost.
def inferno():
    """
    Set the colormap to "inferno".

    This changes the default colormap as well as the colormap of the current
    image if there is one. See ``help(colormaps)`` for more information.
    """
    set_cmap("inferno")

# Autogenerated by boilerplate.py.  Do not edit as changes will be lost.
def plasma():
    """
    Set the colormap to "plasma".

    This changes the default colormap as well as the colormap of the current
    image if there is one. See ``help(colormaps)`` for more information.
    """
    set_cmap("plasma")
```

```
# Autogenerated by boilerplate.py.  Do not edit as changes will be lost.
def viridis():
    """
    Set the colormap to "viridis".

    This changes the default colormap as well as the colormap of the current
    image if there is one. See ``help(colormaps)`` for more information.
    """
    set_cmap("viridis")

# Autogenerated by boilerplate.py.  Do not edit as changes will be lost.
def nipy_spectral():
    """
    Set the colormap to "nipy_spectral".

    This changes the default colormap as well as the colormap of the current
    image if there is one. See ``help(colormaps)`` for more information.
    """
    set_cmap("nipy_spectral")

_setup_pyplot_info_docstrings()
```

https://github.com/scikit-learn/scikit-learn/blob/master/sklearn/preprocessing/_label.py

```
# Authors: Alexandre Gramfort <alexandre.gramfort@inria.fr>
#          Mathieu Blondel <mathieu@mblondel.org>
#          Olivier Grisel <olivier.grisel@ensta.org>
#          Andreas Mueller <amueller@ais.uni-bonn.de>
#          Joel Nothman <joel.nothman@gmail.com>
#          Hamzeh Alsalhi <ha258@cornell.edu>
# License: BSD 3 clause

from collections import defaultdict
import itertools
import array
import warnings

import numpy as np
import scipy.sparse as sp

from ..base import BaseEstimator, TransformerMixin

from ..utils.sparsefuncs import min_max_axis
from ..utils import column_or_1d
from ..utils.validation import check_array
from ..utils.validation import check_is_fitted
from ..utils.validation import _num_samples
from ..utils.validation import _deprecate_positional_args
from ..utils.multiclass import unique_labels
from ..utils.multiclass import type_of_target

__all__ = [
    'label_binarize',
    'LabelBinarizer',
    'LabelEncoder',
    'MultiLabelBinarizer',
]
]

def _encode_numpy(values, uniques=None, encode=False, check_unknown=True):
    # only used in _encode below, see docstring there for details
    if uniques is None:
        if encode:
            uniques, encoded = np.unique(values, return_inverse=True)
            return uniques, encoded
        else:
            # unique sorts
            return np.unique(values)
    if encode:
        if check_unknown:
            diff = _encode_check_unknown(values, uniques)
            if diff:
                raise ValueError("y contains previously unseen labels: %s"
                                 % str(diff))
        encoded = np.searchsorted(uniques, values)
        return uniques, encoded
    else:
        return uniques
```



```

        return res
    else:
        return _encode_numpy(values, uniques, encode,
                             check_unknown=check_unknown)

def _encode_check_unknown(values, uniques, return_mask=False):
    """
    Helper function to check for unknowns in values to be encoded.

    Uses pure python method for object dtype, and numpy method for
    all other dtypes.

    Parameters
    -----
    values : array
        Values to check for unknowns.
    uniques : array
        Allowed uniques values.
    return_mask : bool, default False
        If True, return a mask of the same shape as `values` indicating
        the valid values.

    Returns
    -----
    diff : list
        The unique values present in `values` and not in `uniques` (the
        unknown values).
    valid_mask : boolean array
        Additionally returned if ``return_mask=True``.

    """
    if values.dtype == object:
        uniques_set = set(uniques)
        diff = list(set(values) - uniques_set)
        if return_mask:
            if diff:
                valid_mask = np.array([val in uniques_set for val in values])
            else:
                valid_mask = np.ones(len(values), dtype=bool)
            return diff, valid_mask
        else:
            return diff
    else:
        unique_values = np.unique(values)
        diff = list(np.setdiff1d(unique_values, uniques, assume_unique=True))
        if return_mask:
            if diff:
                valid_mask = np.in1d(values, uniques)
            else:
                valid_mask = np.ones(len(values), dtype=bool)
            return diff, valid_mask
        else:
            return diff

class LabelEncoder(TransformerMixin, BaseEstimator):
    """
    Encode target labels with value between 0 and n_classes-1.

    This transformer should be used to encode target values, *i.e.* `y`, and
    not the input `X`.

    Read more in the :ref:`User Guide <preprocessing_targets>`.

```

```
.. versionadded:: 0.12

Attributes
-----
classes_ : array of shape (n_class,)
    Holds the label for each class.

Examples
-----
`LabelEncoder` can be used to normalize labels.

>>> from sklearn import preprocessing
>>> le = preprocessing.LabelEncoder()
>>> le.fit([1, 2, 2, 6])
LabelEncoder()
>>> le.classes_
array([1, 2, 6])
>>> le.transform([1, 1, 2, 6])
array([0, 0, 1, 2]...)
>>> le.inverse_transform([0, 0, 1, 2])
array([1, 1, 2, 6])

It can also be used to transform non-numerical labels (as long as they are
hashable and comparable) to numerical labels.

>>> le = preprocessing.LabelEncoder()
>>> le.fit(["paris", "paris", "tokyo", "amsterdam"])
LabelEncoder()
>>> list(le.classes_)
['amsterdam', 'paris', 'tokyo']
>>> le.transform(["tokyo", "tokyo", "paris"])
array([2, 2, 1]...)
>>> list(le.inverse_transform([2, 2, 1]))
['tokyo', 'tokyo', 'paris']

See also
-----
sklearn.preprocessing.OrdinalEncoder : Encode categorical features
    using an ordinal encoding scheme.

sklearn.preprocessing.OneHotEncoder : Encode categorical features
    as a one-hot numeric array.

"""

def fit(self, y):
    """Fit label encoder

    Parameters
    -----
    y : array-like of shape (n_samples,)
        Target values.

    Returns
    -----
    self : returns an instance of self.
    """
    y = column_or_1d(y, warn=True)
    self.classes_ = _encode(y)
    return self

def fit_transform(self, y):
    """Fit label encoder and return encoded labels
```

```

Parameters
-----
y : array-like of shape [n_samples]
    Target values.

Returns
-----
y : array-like of shape [n_samples]
"""
y = column_or_1d(y, warn=True)
self.classes_, y = _encode(y, encode=True)
return y

def transform(self, y):
    """Transform labels to normalized encoding.

Parameters
-----
y : array-like of shape [n_samples]
    Target values.

Returns
-----
y : array-like of shape [n_samples]
"""
check_is_fitted(self)
y = column_or_1d(y, warn=True)
# transform of empty array is empty array
if _num_samples(y) == 0:
    return np.array([])

_, y = _encode(y, uniques=self.classes_, encode=True)
return y

def inverse_transform(self, y):
    """Transform labels back to original encoding.

Parameters
-----
y : numpy array of shape [n_samples]
    Target values.

Returns
-----
y : numpy array of shape [n_samples]
"""
check_is_fitted(self)
y = column_or_1d(y, warn=True)
# inverse transform of empty array is empty array
if _num_samples(y) == 0:
    return np.array([])

diff = np.setdiff1d(y, np.arange(len(self.classes_)))
if len(diff):
    raise ValueError(
        "y contains previously unseen labels: %s" % str(diff))
y = np.asarray(y)
return self.classes_[y]

def _more_tags(self):
    return {'X_types': ['1dlabels']}

```

```
class LabelBinarizer(TransformerMixin, BaseEstimator):
    """Binarize labels in a one-vs-all fashion

    Several regression and binary classification algorithms are
    available in scikit-learn. A simple way to extend these algorithms
    to the multi-class classification case is to use the so-called
    one-vs-all scheme.

    At learning time, this simply consists in learning one regressor
    or binary classifier per class. In doing so, one needs to convert
    multi-class labels to binary labels (belong or does not belong
    to the class). LabelBinarizer makes this process easy with the
    transform method.

    At prediction time, one assigns the class for which the corresponding
    model gave the greatest confidence. LabelBinarizer makes this easy
    with the inverse_transform method.

    Read more in the :ref:`User Guide <preprocessing_targets>`.

    Parameters
    -----
    neg_label : int (default: 0)
        Value with which negative labels must be encoded.

    pos_label : int (default: 1)
        Value with which positive labels must be encoded.

    sparse_output : boolean (default: False)
        True if the returned array from transform is desired to be in sparse
        CSR format.

    Attributes
    -----
    classes_ : array of shape [n_class]
        Holds the label for each class.

    y_type_ : str,
        Represents the type of the target data as evaluated by
        utils.multiclass.type_of_target. Possible type are 'continuous',
        'continuous-multioutput', 'binary', 'multiclass',
        'multiclass-multioutput', 'multilabel-indicator', and 'unknown'.

    sparse_input_ : boolean,
        True if the input data to transform is given as a sparse matrix, False
        otherwise.

    Examples
    -----
    >>> from sklearn import preprocessing
    >>> lb = preprocessing.LabelBinarizer()
    >>> lb.fit([1, 2, 6, 4, 2])
    LabelBinarizer()
    >>> lb.classes_
    array([1, 2, 4, 6])
    >>> lb.transform([1, 6])
    array([[1, 0, 0, 0],
           [0, 0, 0, 1]])

    Binary targets transform to a column vector
```

```

>>> lb = preprocessing.LabelBinarizer()
>>> lb.fit_transform(['yes', 'no', 'no', 'yes'])
array([[1],
       [0],
       [0],
       [1]])

Passing a 2D matrix for multilabel classification

>>> import numpy as np
>>> lb.fit(np.array([[0, 1, 1], [1, 0, 0]]))
LabelBinarizer()
>>> lb.classes_
array([0, 1, 2])
>>> lb.transform([0, 1, 2, 1])
array([[1, 0, 0],
       [0, 1, 0],
       [0, 0, 1],
       [0, 1, 0]])

See also
-----
label_binarize : function to perform the transform operation of
    LabelBinarizer with fixed classes.
sklearn.preprocessing.OneHotEncoder : encode categorical features
    using a one-hot aka one-of-K scheme.
"""

 @_deprecate_positional_args
def __init__(self, *, neg_label=0, pos_label=1, sparse_output=False):
    if neg_label >= pos_label:
        raise ValueError("neg_label={0} must be strictly less than "
                         "pos_label={1}.".format(neg_label, pos_label))

    if sparse_output and (pos_label == 0 or neg_label != 0):
        raise ValueError("Sparse binarization is only supported with non "
                         "zero pos_label and zero neg_label, got "
                         "pos_label={0} and neg_label={1}"
                         "".format(pos_label, neg_label))

    self.neg_label = neg_label
    self.pos_label = pos_label
    self.sparse_output = sparse_output

def fit(self, y):
    """Fit label binarizer

    Parameters
    -----
    y : array of shape [n_samples,] or [n_samples, n_classes]
        Target values. The 2-d matrix should only contain 0 and 1,
        represents multilabel classification.

    Returns
    -----
    self : returns an instance of self.
    """
    self.y_type_ = type_of_target(y)
    if 'multioutput' in self.y_type_:
        raise ValueError("Multioutput target data is not supported with "
                         "label binarization")
    if _num_samples(y) == 0:

```

```

        raise ValueError('y has 0 samples: %r' % y)

    self.sparse_input_ = sp.issparse(y)
    self.classes_ = unique_labels(y)
    return self

def fit_transform(self, y):
    """Fit label binarizer and transform multi-class labels to binary
    labels.

    The output of transform is sometimes referred to as
    the 1-of-K coding scheme.

    Parameters
    -----
    y : array or sparse matrix of shape [n_samples,] or \
         [n_samples, n_classes]
        Target values. The 2-d matrix should only contain 0 and 1,
        represents multilabel classification. Sparse matrix can be
        CSR, CSC, COO, DOK, or LIL.

    Returns
    -----
    Y : array or CSR matrix of shape [n_samples, n_classes]
        Shape will be [n_samples, 1] for binary problems.
    """
    return self.fit(y).transform(y)

def transform(self, y):
    """Transform multi-class labels to binary labels

    The output of transform is sometimes referred to by some authors as
    the 1-of-K coding scheme.

    Parameters
    -----
    y : array or sparse matrix of shape [n_samples,] or \
         [n_samples, n_classes]
        Target values. The 2-d matrix should only contain 0 and 1,
        represents multilabel classification. Sparse matrix can be
        CSR, CSC, COO, DOK, or LIL.

    Returns
    -----
    Y : numpy array or CSR matrix of shape [n_samples, n_classes]
        Shape will be [n_samples, 1] for binary problems.
    """
    check_is_fitted(self)

    y_is_multilabel = type_of_target(y).startswith('multilabel')
    if y_is_multilabel and not self.y_type_.startswith('multilabel'):
        raise ValueError("The object was not fitted with multilabel"
                         " input.")

    return label_binarize(y, classes=self.classes_,
                          pos_label=self.pos_label,
                          neg_label=self.neg_label,
                          sparse_output=self.sparse_output)

def inverse_transform(self, Y, threshold=None):
    """Transform binary labels back to multi-class labels

    Parameters

```

```

-----
Y : numpy array or sparse matrix with shape [n_samples, n_classes]
    Target values. All sparse matrices are converted to CSR before
    inverse transformation.

threshold : float or None
    Threshold used in the binary and multi-label cases.

    Use 0 when ``Y`` contains the output of decision_function
    (classifier).
    Use 0.5 when ``Y`` contains the output of predict_proba.

    If None, the threshold is assumed to be half way between
    neg_label and pos_label.

Returns
-----
y : numpy array or CSR matrix of shape [n_samples] Target values.

Notes
-----
In the case when the binary labels are fractional
(probabilistic), inverse_transform chooses the class with the
greatest value. Typically, this allows to use the output of a
linear model's decision_function method directly as the input
of inverse_transform.

"""
check_is_fitted(self)

if threshold is None:
    threshold = (self.pos_label + self.neg_label) / 2.

if self.y_type_ == "multiclass":
    y_inv = _inverse_binarize_multiclass(Y, self.classes_)
else:
    y_inv = _inverse_binarize_thresholding(Y, self.y_type_,
                                             self.classes_, threshold)

if self.sparse_input_:
    y_inv = sp.csr_matrix(y_inv)
elif sp.issparse(y_inv):
    y_inv = y_inv.toarray()

return y_inv

def _more_tags(self):
    return {'X_types': ['1dlabels']}

```

`@_deprecate_positional_args`

```

def label_binarize(y, *, classes, neg_label=0, pos_label=1,
                   sparse_output=False):
    """Binarize labels in a one-vs-all fashion

    Several regression and binary classification algorithms are
    available in scikit-learn. A simple way to extend these algorithms
    to the multi-class classification case is to use the so-called
    one-vs-all scheme.

    This function makes it possible to compute this transformation for a
    fixed set of class labels known ahead of time.

Parameters

```

```
-----  
y : array-like  
    Sequence of integer labels or multilabel data to encode.  
  
classes : array-like of shape [n_classes]  
    Uniquely holds the label for each class.  
  
neg_label : int (default: 0)  
    Value with which negative labels must be encoded.  
  
pos_label : int (default: 1)  
    Value with which positive labels must be encoded.  
  
sparse_output : boolean (default: False),  
    Set to true if output binary array is desired in CSR sparse format
```

Returns

```
-----  
Y : numpy array or CSR matrix of shape [n_samples, n_classes]  
    Shape will be [n_samples, 1] for binary problems.
```

Examples

```
>>> from sklearn.preprocessing import label_binarize  
>>> label_binarize([1, 6], classes=[1, 2, 4, 6])  
array([[1, 0, 0, 0],  
      [0, 0, 0, 1]])
```

The class ordering is preserved:

```
>>> label_binarize([1, 6], classes=[1, 6, 4, 2])  
array([[1, 0, 0, 0],  
      [0, 1, 0, 0]])
```

Binary targets transform to a column vector

```
>>> label_binarize(['yes', 'no', 'no', 'yes'], classes=['no', 'yes'])  
array([[1],  
      [0],  
      [0],  
      [1]])
```

See also

```
-----  
LabelBinarizer : class used to wrap the functionality of label_binarize and  
    allow for fitting to classes independently of the transform operation
```

```
"""  
if not isinstance(y, list):  
    # XXX Workaround that will be removed when list of list format is  
    # dropped  
    y = check_array(y, accept_sparse='csr', ensure_2d=False, dtype=None)  
else:  
    if _num_samples(y) == 0:  
        raise ValueError('y has 0 samples: %r' % y)  
if neg_label >= pos_label:  
    raise ValueError("neg_label={0} must be strictly less than "  
                     "pos_label={1}.".format(neg_label, pos_label))  
  
if (sparse_output and (pos_label == 0 or neg_label != 0)):  
    raise ValueError("Sparse binarization is only supported with non "  
                     "zero pos_label and zero neg_label, got "  
                     "pos_label={0} and neg_label={1}"  
                     ".format(pos_label, neg_label))
```

```

# To account for pos_label == 0 in the dense case
pos_switch = pos_label == 0
if pos_switch:
    pos_label = -neg_label

y_type = type_of_target(y)
if 'multioutput' in y_type:
    raise ValueError("Multioutput target data is not supported with label "
                     "binarization")
if y_type == 'unknown':
    raise ValueError("The type of target data is not known")

n_samples = y.shape[0] if sp.issparse(y) else len(y)
n_classes = len(classes)
classes = np.asarray(classes)

if y_type == "binary":
    if n_classes == 1:
        if sparse_output:
            return sp.csr_matrix((n_samples, 1), dtype=int)
        else:
            Y = np.zeros((len(y), 1), dtype=np.int)
            Y += neg_label
            return Y
    elif len(classes) >= 3:
        y_type = "multiclass"

sorted_class = np.sort(classes)
if y_type == "multilabel-indicator":
    y_n_classes = y.shape[1] if hasattr(y, 'shape') else len(y[0])
    if classes.size != y_n_classes:
        raise ValueError("classes {0} mismatch with the labels {1}"
                         " found in the data"
                         .format(classes, unique_labels(y)))

if y_type in ("binary", "multiclass"):
    y = column_or_1d(y)

    # pick out the known labels from y
    y_in_classes = np.in1d(y, classes)
    y_seen = y[y_in_classes]
    indices = np.searchsorted(sorted_class, y_seen)
    indptr = np.hstack((0, np.cumsum(y_in_classes)))

    data = np.empty_like(indices)
    data.fill(pos_label)
    Y = sp.csr_matrix((data, indices, indptr),
                      shape=(n_samples, n_classes))
elif y_type == "multilabel-indicator":
    Y = sp.csr_matrix(y)
    if pos_label != 1:
        data = np.empty_like(Y.data)
        data.fill(pos_label)
        Y.data = data
else:
    raise ValueError("%s target data is not supported with label "
                     "binarization" % y_type)

if not sparse_output:
    Y = Y.toarray()
    Y = Y.astype(int, copy=False)

```

```

    if neg_label != 0:
        Y[Y == 0] = neg_label

    if pos_switch:
        Y[Y == pos_label] = 0
else:
    Y.data = Y.data.astype(int, copy=False)

# preserve label ordering
if np.any(classes != sorted_class):
    indices = np.searchsorted(sorted_class, classes)
    Y = Y[:, indices]

if y_type == "binary":
    if sparse_output:
        Y = Y.getcol(-1)
    else:
        Y = Y[:, -1].reshape((-1, 1))

return Y

def _inverse_binarize_multiclass(y, classes):
    """Inverse label binarization transformation for multiclass.

    Multiclass uses the maximal score instead of a threshold.
    """
    classes = np.asarray(classes)

    if sp.issparse(y):
        # Find the argmax for each row in y where y is a CSR matrix

        y = y.tocsr()
        n_samples, n_outputs = y.shape
        outputs = np.arange(n_outputs)
        row_max = min_max_axis(y, 1)[1]
        row_nnz = np.diff(y.indptr)

        y_data_repeated_max = np.repeat(row_max, row_nnz)
        # picks out all indices obtaining the maximum per row
        y_i_all_argmax = np.flatnonzero(y_data_repeated_max == y.data)

        # For corner case where last row has a max of 0
        if row_max[-1] == 0:
            y_i_all_argmax = np.append(y_i_all_argmax, [len(y.data)])

        # Gets the index of the first argmax in each row from y_i_all_argmax
        index_first_argmax = np.searchsorted(y_i_all_argmax, y.indptr[:-1])
        # first argmax of each row
        y_ind_ext = np.append(y.indices, [0])
        y_i_argmax = y_ind_ext[y_i_all_argmax[index_first_argmax]]
        # Handle rows of all 0
        y_i_argmax[np.where(row_nnz == 0)[0]] = 0

        # Handles rows with max of 0 that contain negative numbers
        samples = np.arange(n_samples)[(row_nnz > 0) &
                                      (row_max.ravel() == 0)]
        for i in samples:
            ind = y.indices[y.indptr[i]:y.indptr[i + 1]]
            y_i_argmax[i] = classes[np.setdiff1d(outputs, ind)][0]

        return classes[y_i_argmax]
    else:

```

```

        return classes.take(y.argmax(axis=1), mode="clip")

def _inverse_binarize_thresholding(y, output_type, classes, threshold):
    """Inverse label binarization transformation using thresholding."""

    if output_type == "binary" and y.ndim == 2 and y.shape[1] > 2:
        raise ValueError("output_type='binary', but y.shape = {0}.".format(y.shape))

    if output_type != "binary" and y.shape[1] != len(classes):
        raise ValueError("The number of class is not equal to the number of "
                         "dimension of y.")

    classes = np.asarray(classes)

    # Perform thresholding
    if sp.issparse(y):
        if threshold > 0:
            if y.format not in ('csr', 'csc'):
                y = y.tocsr()
            y.data = np.array(y.data > threshold, dtype=np.int)
            y.eliminate_zeros()
        else:
            y = np.array(y.toarray() > threshold, dtype=np.int)
    else:
        y = np.array(y > threshold, dtype=np.int)

    # Inverse transform data
    if output_type == "binary":
        if sp.issparse(y):
            y = y.toarray()
        if y.ndim == 2 and y.shape[1] == 2:
            return classes[y[:, 1]]
        else:
            if len(classes) == 1:
                return np.repeat(classes[0], len(y))
            else:
                return classes[y.ravel()]
    elif output_type == "multilabel-indicator":
        return y
    else:
        raise ValueError("{0} format is not supported".format(output_type))

class MultiLabelBinarizer(TransformerMixin, BaseEstimator):
    """Transform between iterable of iterables and a multilabel format

    Although a list of sets or tuples is a very intuitive format for multilabel
    data, it is unwieldy to process. This transformer converts between this
    intuitive format and the supported multilabel format: a (samples x classes)
    binary matrix indicating the presence of a class label.

    Parameters
    -----
    classes : array-like of shape [n_classes] (optional)
        Indicates an ordering for the class labels.
        All entries should be unique (cannot contain duplicate classes).

    sparse_output : boolean (default: False),
        Set to true if output binary array is desired in CSR sparse format

```

```
Attributes
-----
classes_ : array of labels
    A copy of the `classes` parameter where provided,
    or otherwise, the sorted set of classes found when fitting.
```

Examples

```
>>> from sklearn.preprocessing import MultiLabelBinarizer
>>> mlb = MultiLabelBinarizer()
>>> mlb.fit_transform([(1, 2), (3,)])
array([[1, 1, 0],
       [0, 0, 1]])
>>> mlb.classes_
array([1, 2, 3])

>>> mlb.fit_transform([('sci-fi', 'thriller'), ('comedy')])
array([[0, 1, 1],
       [1, 0, 0]])
>>> list(mlb.classes_)
['comedy', 'sci-fi', 'thriller']
```

A common mistake is to pass in a list, which leads to the following issue:

```
>>> mlb = MultiLabelBinarizer()
>>> mlb.fit(['sci-fi', 'thriller', 'comedy'])
MultiLabelBinarizer()
>>> mlb.classes_
array(['-', 'c', 'd', 'e', 'f', 'h', 'i', 'l', 'm', 'o', 'r', 's', 't',
       'y'], dtype=object)
```

To correct this, the list of labels should be passed in as:

```
>>> mlb = MultiLabelBinarizer()
>>> mlb.fit([('sci-fi', 'thriller', 'comedy')])
MultiLabelBinarizer()
>>> mlb.classes_
array(['comedy', 'sci-fi', 'thriller'], dtype=object)
```

See also

```
sklearn.preprocessing.OneHotEncoder : encode categorical features
    using a one-hot aka one-of-K scheme.
```

"""

```
@_deprecate_positional_args
def __init__(self, *, classes=None, sparse_output=False):
    self.classes = classes
    self.sparse_output = sparse_output

def fit(self, y):
    """Fit the label sets binarizer, storing :term:`classes_`
```

Parameters

y : iterable of iterables

A set of labels (any orderable and hashable object) for each sample. If the `classes` parameter is set, `y` will not be iterated.

Returns

```

    self : returns this MultiLabelBinarizer instance
    """
    self._cached_dict = None
    if self.classes is None:
        classes = sorted(set(itertools.chain.from_iterable(y)))
    elif len(set(self.classes)) < len(self.classes):
        raise ValueError("The classes argument contains duplicate "
                         "classes. Remove these duplicates before passing "
                         "them to MultiLabelBinarizer.")
    else:
        classes = self.classes
    dtype = np.int if all(isinstance(c, int) for c in classes) else object
    self.classes_ = np.empty(len(classes), dtype=dtype)
    self.classes_[:] = classes
    return self

def fit_transform(self, y):
    """Fit the label sets binarizer and transform the given label sets

    Parameters
    -----
    y : iterable of iterables
        A set of labels (any orderable and hashable object) for each
        sample. If the `classes` parameter is set, `y` will not be
        iterated.

    Returns
    -----
    y_indicator : array or CSR matrix, shape (n_samples, n_classes)
        A matrix such that `y_indicator[i, j] = 1` iff `classes_[j]` is in
        `y[i]`, and 0 otherwise.
    """
    self._cached_dict = None

    if self.classes is not None:
        return self.fit(y).transform(y)

    # Automatically increment on new class
    class_mapping = defaultdict(int)
    class_mapping.default_factory = class_mapping.__len__
    yt = self._transform(y, class_mapping)

    # sort classes and reorder columns
    tmp = sorted(class_mapping, key=class_mapping.get)

    # (make safe for tuples)
    dtype = np.int if all(isinstance(c, int) for c in tmp) else object
    class_mapping = np.empty(len(tmp), dtype=dtype)
    class_mapping[:] = tmp
    self.classes_, inverse = np.unique(class_mapping, return_inverse=True)
    # ensure yt.indices keeps its current dtype
    yt.indices = np.array(inverse[xt.indices], dtype=yt.indices.dtype,
                          copy=False)

    if not self.sparse_output:
        yt = yt.toarray()

    return yt

def transform(self, y):
    """Transform the given label sets

    Parameters

```

```

-----
y : iterable of iterables
    A set of labels (any orderable and hashable object) for each
    sample. If the `classes` parameter is set, `y` will not be
    iterated.

Returns
-----
y_indicator : array or CSR matrix, shape (n_samples, n_classes)
    A matrix such that `y_indicator[i, j] = 1` iff `classes_[j]` is in
    `y[i]`, and 0 otherwise.
"""
check_is_fitted(self)

class_to_index = self._build_cache()
yt = self._transform(y, class_to_index)

if not self.sparse_output:
    yt = yt.toarray()

return yt

def _build_cache(self):
    if self._cached_dict is None:
        self._cached_dict = dict(zip(self.classes_,
                                      range(len(self.classes_))))

    return self._cached_dict

def _transform(self, y, class_mapping):
    """Transforms the label sets with a given mapping

Parameters
-----
y : iterable of iterables
class_mapping : Mapping
    Maps from label to column index in label indicator matrix

Returns
-----
y_indicator : sparse CSR matrix, shape (n_samples, n_classes)
    Label indicator matrix
"""

    indices = array.array('i')
    indptr = array.array('i', [0])
    unknown = set()
    for labels in y:
        index = set()
        for label in labels:
            try:
                index.add(class_mapping[label])
            except KeyError:
                unknown.add(label)
        indices.extend(index)
        indptr.append(len(indices))
    if unknown:
        warnings.warn('unknown class(es) {0} will be ignored'
                      .format(sorted(unknown, key=str)))
    data = np.ones(len(indices), dtype=int)

    return sp.csr_matrix((data, indices, indptr),
                         shape=(len(indptr) - 1, len(class_mapping)))

```

```
def inverse_transform(self, yt):
    """Transform the given indicator matrix into label sets

    Parameters
    -----
    yt : array or sparse matrix of shape (n_samples, n_classes)
        A matrix containing only 1s and 0s.

    Returns
    -----
    y : list of tuples
        The set of labels for each sample such that `y[i]` consists of
        `classes_[j]` for each `yt[i, j] == 1`.
    """
    check_is_fitted(self)

    if yt.shape[1] != len(self.classes_):
        raise ValueError('Expected indicator for {} classes, but got {}'.format(len(self.classes_), yt.shape[1]))

    if sp.issparse(yt):
        yt = yt.tocsr()
        if len(yt.data) != 0 and len(np.setdiff1d(yt.data, [0, 1])) > 0:
            raise ValueError('Expected only 0s and 1s in label indicator.')
        return [tuple(self.classes_.take(yt.indices[start:end])) for start, end in zip(yt.indptr[:-1], yt.indptr[1:])]

    else:
        unexpected = np.setdiff1d(yt, [0, 1])
        if len(unexpected) > 0:
            raise ValueError('Expected only 0s and 1s in label indicator. '
                             'Also got {}'.format(unexpected))
        return [tuple(self.classes_.compress(indicators)) for indicators
                in yt]

def _more_tags(self):
    return {'X_types': ['2dlabels']}
```

<https://github.com/mwaskom/seaborn/blob/master/seaborn/categorical.py>

```
from textwrap import dedent
import colorsys
import numpy as np
from scipy import stats
import pandas as pd
import matplotlib as mpl
from matplotlib.collections import PatchCollection
import matplotlib.patches as Patches
import matplotlib.pyplot as plt
import warnings
from distutils.version import LooseVersion

from . import utils
from .utils import iqr, categorical_order, remove_na
from .algorithms import bootstrap
from .palettes import color_palette, husl_palette, light_palette, dark_palette
from .axisgrid import FacetGrid, _facet_docs

__all__ = [
    "catplot", "factorplot",
    "stripplot", "swarmplot",
    "boxplot", "violinplot", "boxenplot", "lvplot",
    "pointplot", "barplot", "countplot",
]
]

class _CategoricalPlotter(object):

    width = .8
    default_palette = "light"

    def establish_variables(self, x=None, y=None, hue=None, data=None,
                           orient=None, order=None, hue_order=None,
                           units=None):
        """Convert input specification into a common representation."""
        # Option 1:
        # We are plotting a wide-form dataset
        # -----
        if x is None and y is None:

            # Do a sanity check on the inputs
            if hue is not None:
                error = "Cannot use `hue` without `x` or `y`"
                raise ValueError(error)

            # No hue grouping with wide inputs
            plot_hues = None
            hue_title = None
            hue_names = None

            # No statistical units with wide inputs
            plot_units = None

            # We also won't get axes labels here
```

```

value_label = None
group_label = None

# Option 1a:
# The input data is a Pandas DataFrame
# ----

if isinstance(data, pd.DataFrame):

    # Order the data correctly
    if order is None:
        order = []
        # Reduce to just numeric columns
        for col in data:
            try:
                data[col].astype(np.float)
                order.append(col)
            except ValueError:
                pass
    plot_data = data[order]
    group_names = order
    group_label = data.columns.name

    # Convert to a list of arrays, the common representation
    iter_data = plot_data.iteritems()
    plot_data = [np.asarray(s, np.float) for k, s in iter_data]

# Option 1b:
# The input data is an array or list
# ----

else:

    # We can't reorder the data
    if order is not None:
        error = "Input data must be a pandas object to reorder"
        raise ValueError(error)

    # The input data is an array
    if hasattr(data, "shape"):
        if len(data.shape) == 1:
            if np.isscalar(data[0]):
                plot_data = [data]
            else:
                plot_data = list(data)
        elif len(data.shape) == 2:
            nr, nc = data.shape
            if nr == 1 or nc == 1:
                plot_data = [data.ravel()]
            else:
                plot_data = [data[:, i] for i in range(nc)]
        else:
            error = ("Input `data` can have no "
                    "more than 2 dimensions")
            raise ValueError(error)

    # Check if `data` is None to let us bail out here (for testing)
    elif data is None:
        plot_data = [[]]

    # The input data is a flat list
    elif np.isscalar(data[0]):
        plot_data = [data]

```

```

# The input data is a nested list
# This will catch some things that might fail later
# but exhaustive checks are hard
else:
    plot_data = data

# Convert to a list of arrays, the common representation
plot_data = [np.asarray(d, np.float) for d in plot_data]

# The group names will just be numeric indices
group_names = list(range((len(plot_data)))))

# Figure out the plotting orientation
orient = "h" if str(orient).startswith("h") else "v"

# Option 2:
# We are plotting a long-form dataset
# -----
else:

    # See if we need to get variables from `data`
    if data is not None:
        x = data.get(x, x)
        y = data.get(y, y)
        hue = data.get(hue, hue)
        units = data.get(units, units)

    # Validate the inputs
    for var in [x, y, hue, units]:
        if isinstance(var, str):
            err = "Could not interpret input '{}'".format(var)
            raise ValueError(err)

    # Figure out the plotting orientation
    orient = self.infer_orient(x, y, orient)

    # Option 2a:
    # We are plotting a single set of data
    # -----
    if x is None or y is None:

        # Determine where the data are
        vals = y if x is None else x

        # Put them into the common representation
        plot_data = [np.asarray(vals)]

        # Get a label for the value axis
        if hasattr(vals, "name"):
            value_label = vals.name
        else:
            value_label = None

    # This plot will not have group labels or hue nesting
    groups = None
    group_label = None
    group_names = []
    plot_hues = None
    hue_names = None
    hue_title = None
    plot_units = None

```

```

# Option 2b:
# We are grouping the data values by another variable
# -----
else:

    # Determine which role each variable will play
    if orient == "v":
        vals, groups = y, x
    else:
        vals, groups = x, y

    # Get the categorical axis label
    group_label = None
    if hasattr(groups, "name"):
        group_label = groups.name

    # Get the order on the categorical axis
    group_names = categorical_order(groups, order)

    # Group the numeric data
    plot_data, value_label = self._group_longform(vals, groups,
                                                   group_names)

    # Now handle the hue levels for nested ordering
    if hue is None:
        plot_hues = None
        hue_title = None
        hue_names = None
    else:

        # Get the order of the hue levels
        hue_names = categorical_order(hue, hue_order)

        # Group the hue data
        plot_hues, hue_title = self._group_longform(hue, groups,
                                                      group_names)

    # Now handle the units for nested observations
    if units is None:
        plot_units = None
    else:
        plot_units, _ = self._group_longform(units, groups,
                                              group_names)

# Assign object attributes
# -----
self.orientation = orient
self.plot_data = plot_data
self.group_label = group_label
self.value_label = value_label
self.group_names = group_names
self.plot_hues = plot_hues
self.hue_title = hue_title
self.hue_names = hue_names
self.plot_units = plot_units

def _group_longform(self, vals, grouper, order):
    """Group a long-form variable by another with correct order."""
    # Ensure that the groupby will work
    if not isinstance(vals, pd.Series):
        if isinstance(grouper, pd.Series):
            index = grouper.index

```

```

        else:
            index = None
            vals = pd.Series(vals, index=index)

        # Group the val data
        grouped_vals = vals.groupby(grouper)
        out_data = []
        for g in order:
            try:
                g_vals = grouped_vals.get_group(g)
            except KeyError:
                g_vals = np.array([])
            out_data.append(g_vals)

    # Get the vals axis label
    label = vals.name

    return out_data, label

def establish_colors(self, color, palette, saturation):
    """Get a list of colors for the main component of the plots."""
    if self.hue_names is None:
        n_colors = len(self.plot_data)
    else:
        n_colors = len(self.hue_names)

    # Determine the main colors
    if color is None and palette is None:
        # Determine whether the current palette will have enough values
        # If not, we'll default to the husl palette so each is distinct
        current_palette = utils.get_color_cycle()
        if n_colors <= len(current_palette):
            colors = color_palette(n_colors=n_colors)
        else:
            colors = husl_palette(n_colors, l=.7) # noqa

    elif palette is None:
        # When passing a specific color, the interpretation depends
        # on whether there is a hue variable or not.
        # If so, we will make a blend palette so that the different
        # levels have some amount of variation.
        if self.hue_names is None:
            colors = [color] * n_colors
        else:
            if self.default_palette == "light":
                colors = light_palette(color, n_colors)
            elif self.default_palette == "dark":
                colors = dark_palette(color, n_colors)
            else:
                raise RuntimeError("No default palette specified")
    else:
        # Let `palette` be a dict mapping level to color
        if isinstance(palette, dict):
            if self.hue_names is None:
                levels = self.group_names
            else:
                levels = self.hue_names
            palette = [palette[l] for l in levels]

        colors = color_palette(palette, n_colors)

    # Desaturate a bit because these are patches

```

```

        if saturation < 1:
            colors = color_palette(colors, desat=saturation)

        # Convert the colors to a common representations
        rgb_colors = color_palette(colors)

        # Determine the gray color to use for the lines framing the plot
        light_vals = [colorsys.rgb_to_hls(*c)[1] for c in rgb_colors]
        lum = min(light_vals) * .6
        gray = mpl.colors.rgb2hex((lum, lum, lum))

        # Assign object attributes
        self.colors = rgb_colors
        self.gray = gray

    def infer_orient(self, x, y, orient=None):
        """Determine how the plot should be oriented based on the data."""
        orient = str(orient)

        def is_categorical(s):
            return pd.api.types.is_categorical_dtype(s)

        def is_not_numeric(s):
            try:
                np.asarray(s, dtype=np.float)
            except ValueError:
                return True
            return False

        no_numeric = "Neither the `x` nor `y` variable appears to be numeric."

        if orient.startswith("v"):
            return "v"
        elif orient.startswith("h"):
            return "h"
        elif x is None:
            return "v"
        elif y is None:
            return "h"
        elif is_categorical(y):
            if is_categorical(x):
                raise ValueError(no_numeric)
            else:
                return "h"
        elif is_not_numeric(y):
            if is_not_numeric(x):
                raise ValueError(no_numeric)
            else:
                return "h"
        else:
            return "v"

    @property
    def hue_offsets(self):
        """A list of center positions for plots when hue nesting is used."""
        n_levels = len(self.hue_names)
        if self.dodge:
            each_width = self.width / n_levels
            offsets = np.linspace(0, self.width - each_width, n_levels)
            offsets -= offsets.mean()
        else:
            offsets = np.zeros(n_levels)

```

```

    return offsets

@property
def nested_width(self):
    """A float with the width of plot elements when hue nesting is used."""
    if self.dodge:
        width = self.width / len(self.hue_names) * .98
    else:
        width = self.width
    return width

def annotate_axes(self, ax):
    """Add descriptive labels to an Axes object."""
    if self.orient == "v":
        xlabel, ylabel = self.group_label, self.value_label
    else:
        xlabel, ylabel = self.value_label, self.group_label

    if xlabel is not None:
        ax.set_xlabel(xlabel)
    if ylabel is not None:
        ax.set_ylabel(ylabel)

    if self.orient == "v":
        ax.set_xticks(np.arange(len(self.plot_data)))
        ax.set_xticklabels(self.group_names)
    else:
        ax.set_yticks(np.arange(len(self.plot_data)))
        ax.set_yticklabels(self.group_names)

    if self.orient == "v":
        ax.xaxis.grid(False)
        ax.set_xlim(-.5, len(self.plot_data) - .5, auto=None)
    else:
        ax.yaxis.grid(False)
        ax.set_ylim(-.5, len(self.plot_data) - .5, auto=None)

    if self.hue_names is not None:
        leg = ax.legend(loc="best", title=self.hue_title)
        if self.hue_title is not None:
            if LooseVersion(mpl.__version__) < "3.0":
                # Old Matplotlib has no legend title size rcpParam
                try:
                    title_size = mpl.rcParams["axes.labelsize"] * .85
                except TypeError: # labelsize is something like "large"
                    title_size = mpl.rcParams["axes.labelsize"]
                prop = mpl.font_manager.FontProperties(size=title_size)
                leg.set_title(self.hue_title, prop=prop)

def add_legend_data(self, ax, color, label):
    """Add a dummy patch object so we can get legend data."""
    rect = plt.Rectangle([0, 0], 0, 0,
                        linewidth=self.linewidth / 2,
                        edgecolor=self.gray,
                        facecolor=color,
                        label=label)
    ax.add_patch(rect)

class _BoxPlotter(_CategoricalPlotter):

    def __init__(self, x, y, hue, data, order, hue_order,
                 orient, color, palette, saturation,

```

```

        width, dodge, fliersize, linewidth):

    self._establish_variables(x, y, hue, data, orient, order, hue_order)
    self._establish_colors(color, palette, saturation)

    self.dodge = dodge
    self.width = width
    self.fliersize = fliersize

    if linewidth is None:
        linewidth = mpl.rcParams["lines.linewidth"]
    self.linewidth = linewidth

def draw_boxplot(self, ax, kws):
    """Use matplotlib to draw a boxplot on an Axes."""
    vert = self.orient == "v"

    props = {}
    for obj in ["box", "whisker", "cap", "median", "flier"]:
        props[obj] = kws.pop(obj + "props", {})

    for i, group_data in enumerate(self.plot_data):

        if self.plot_hues is None:

            # Handle case where there is data at this level
            if group_data.size == 0:
                continue

            # Draw a single box or a set of boxes
            # with a single level of grouping
            box_data = np.asarray(remove_na(group_data))

            # Handle case where there is no non-null data
            if box_data.size == 0:
                continue

            artist_dict = ax.boxplot(box_data,
                                      vert=vert,
                                      patch_artist=True,
                                      positions=[i],
                                      widths=self.width,
                                      **kws)

            color = self.colors[i]
            self._restyle_boxplot(artist_dict, color, props)
        else:
            # Draw nested groups of boxes
            offsets = self.hue_offsets
            for j, hue_level in enumerate(self.hue_names):

                # Add a legend for this hue level
                if not i:
                    self._add_legend_data(ax, self.colors[j], hue_level)

                # Handle case where there is data at this level
                if group_data.size == 0:
                    continue

                hue_mask = self.plot_hues[i] == hue_level
                box_data = np.asarray(remove_na(group_data[hue_mask]))

                # Handle case where there is no non-null data
                if box_data.size == 0:

```

```

        continue

        center = i + offsets[j]
        artist_dict = ax.boxplot(box_data,
                                  vert=vert,
                                  patch_artist=True,
                                  positions=[center],
                                  widths=self.nested_width,
                                  **kws)
        self.restyle_boxplot(artist_dict, self.colors[j], props)
        # Add legend data, but just for one set of boxes

def restyle_boxplot(self, artist_dict, color, props):
    """Take a drawn matplotlib boxplot and make it look nice."""
    for box in artist_dict["boxes"]:
        box.update(dict(facecolor=color,
                        zorder=.9,
                        edgecolor=self.gray,
                        linewidth=self.linewidth))
    box.update(props["box"])
    for whisk in artist_dict["whiskers"]:
        whisk.update(dict(color=self.gray,
                           linewidth=self.linewidth,
                           linestyle="-"))
    whisk.update(props["whisker"])
    for cap in artist_dict["caps"]:
        cap.update(dict(color=self.gray,
                        linewidth=self.linewidth))
    cap.update(props["cap"])
    for med in artist_dict["medians"]:
        med.update(dict(color=self.gray,
                        linewidth=self.linewidth))
    med.update(props["median"])
    for fly in artist_dict["fliers"]:
        fly.update(dict(markerfacecolor=self.gray,
                        marker="d",
                        markeredgecolor=self.gray,
                        markersize=self.fliersize))
    fly.update(props["flier"])

def plot(self, ax, boxplot_kws):
    """Make the plot."""
    self.draw_boxplot(ax, boxplot_kws)
    self.annotate_axes(ax)
    if self.orient == "h":
        ax.invert_yaxis()

class _ViolinPlotter(_CategoricalPlotter):

    def __init__(self, x, y, hue, data, order, hue_order,
                 bw, cut, scale, scale_hue, gridsize,
                 width, inner, split, dodge, orient, linewidth,
                 color, palette, saturation):

        self.establish_variables(x, y, hue, data, orient, order, hue_order)
        self.establish_colors(color, palette, saturation)
        self.estimate_densities(bw, cut, scale, scale_hue, gridsize)

        self.gridsize = gridsize
        self.width = width
        self.dodge = dodge

```

```

if inner is not None:
    if not any([inner.startswith("quart"),
               inner.startswith("box"),
               inner.startswith("stick"),
               inner.startswith("point")]):
        err = "Inner style '{}' not recognized".format(inner)
        raise ValueError(err)
    self.inner = inner

if split and self.hue_names is not None and len(self.hue_names) != 2:
    msg = "There must be exactly two hue levels to use `split`."
    raise ValueError(msg)
self.split = split

if linewidth is None:
    linewidth = mpl.rcParams["lines.linewidth"]
self.linewidth = linewidth

def estimate_densities(self, bw, cut, scale, scale_hue, gridsize):
    """Find the support and density for all of the data."""
    # Initialize data structures to keep track of plotting data
    if self.hue_names is None:
        support = []
        density = []
        counts = np.zeros(len(self.plot_data))
        max_density = np.zeros(len(self.plot_data))
    else:
        support = [[] for _ in self.plot_data]
        density = [[] for _ in self.plot_data]
        size = len(self.group_names), len(self.hue_names)
        counts = np.zeros(size)
        max_density = np.zeros(size)

    for i, group_data in enumerate(self.plot_data):

        # Option 1: we have a single level of grouping
        # -----
        if self.plot_hues is None:

            # Strip missing datapoints
            kde_data = remove_na(group_data)

            # Handle special case of no data at this level
            if kde_data.size == 0:
                support.append(np.array([]))
                density.append(np.array([1.]))
                counts[i] = 0
                max_density[i] = 0
                continue

            # Handle special case of a single unique datapoint
            elif np.unique(kde_data).size == 1:
                support.append(np.unique(kde_data))
                density.append(np.array([1.]))
                counts[i] = 1
                max_density[i] = 0
                continue

            # Fit the KDE and get the used bandwidth size
            kde, bw_used = self.fit_kde(kde_data, bw)

        # Determine the support grid and get the density over it

```

```

support_i = self.kde_support(kde_data, bw_used, cut, gridsize)
density_i = kde.evaluate(support_i)

# Update the data structures with these results
support.append(support_i)
density.append(density_i)
counts[i] = kde_data.size
max_density[i] = density_i.max()

# Option 2: we have nested grouping by a hue variable
# -----
else:
    for j, hue_level in enumerate(self.hue_names):

        # Handle special case of no data at this category level
        if not group_data.size:
            support[i].append(np.array([]))
            density[i].append(np.array([1.]))
            counts[i, j] = 0
            max_density[i, j] = 0
            continue

        # Select out the observations for this hue level
        hue_mask = self.plot_hues[i] == hue_level

        # Strip missing datapoints
        kde_data = remove_na(group_data[hue_mask])

        # Handle special case of no data at this level
        if kde_data.size == 0:
            support[i].append(np.array([]))
            density[i].append(np.array([1.]))
            counts[i, j] = 0
            max_density[i, j] = 0
            continue

        # Handle special case of a single unique datapoint
        elif np.unique(kde_data).size == 1:
            support[i].append(np.unique(kde_data))
            density[i].append(np.array([1.]))
            counts[i, j] = 1
            max_density[i, j] = 0
            continue

        # Fit the KDE and get the used bandwidth size
        kde, bw_used = self.fit_kde(kde_data, bw)

        # Determine the support grid and get the density over it
        support_ij = self.kde_support(kde_data, bw_used,
                                       cut, gridsize)
        density_ij = kde.evaluate(support_ij)

        # Update the data structures with these results
        support[i].append(support_ij)
        density[i].append(density_ij)
        counts[i, j] = kde_data.size
        max_density[i, j] = density_ij.max()

    # Scale the height of the density curve.
    # For a violinplot the density is non-quantitative.
    # The objective here is to scale the curves relative to 1 so that
    # they can be multiplied by the width parameter during plotting.

```

```

    if scale == "area":
        self.scale_area(density, max_density, scale_hue)

    elif scale == "width":
        self.scale_width(density)

    elif scale == "count":
        self.scale_count(density, counts, scale_hue)

    else:
        raise ValueError("scale method '{}' not recognized".format(scale))

    # Set object attributes that will be used while plotting
    self.support = support
    self.density = density

def fit_kde(self, x, bw):
    """Estimate a KDE for a vector of data with flexible bandwidth."""
    kde = stats.gaussian_kde(x, bw)

    # Extract the numeric bandwidth from the KDE object
    bw_used = kde.factor

    # At this point, bw will be a numeric scale factor.
    # To get the actual bandwidth of the kernel, we multiple by the
    # unbiased standard deviation of the data, which we will use
    # elsewhere to compute the range of the support.
    bw_used = bw_used * x.std(ddof=1)

    return kde, bw_used

def kde_support(self, x, bw, cut, gridsize):
    """Define a grid of support for the violin."""
    support_min = x.min() - bw * cut
    support_max = x.max() + bw * cut
    return np.linspace(support_min, support_max, gridsize)

def scale_area(self, density, max_density, scale_hue):
    """Scale the relative area under the KDE curve.

    This essentially preserves the "standard" KDE scaling, but the
    resulting maximum density will be 1 so that the curve can be
    properly multiplied by the violin width.

    """
    if self.hue_names is None:
        for d in density:
            if d.size > 1:
                d /= max_density.max()
    else:
        for i, group in enumerate(density):
            for d in group:
                if scale_hue:
                    max = max_density[i].max()
                else:
                    max = max_density.max()
                if d.size > 1:
                    d /= max

def scale_width(self, density):
    """Scale each density curve to the same height."""
    if self.hue_names is None:

```

```

        for d in density:
            d /= d.max()
    else:
        for group in density:
            for d in group:
                d /= d.max()

    def scale_count(self, density, counts, scale_hue):
        """Scale each density curve by the number of observations."""
        if self.hue_names is None:
            if counts.max() == 0:
                d = 0
            else:
                for count, d in zip(counts, density):
                    d /= d.max()
                    d *= count / counts.max()
        else:
            for i, group in enumerate(density):
                for j, d in enumerate(group):
                    if counts[i].max() == 0:
                        d = 0
                    else:
                        count = counts[i, j]
                        if scale_hue:
                            scaler = count / counts[i].max()
                        else:
                            scaler = count / counts.max()
                        d /= d.max()
                        d *= scaler

    @property
    def dwidth(self):

        if self.hue_names is None or not self.dodge:
            return self.width / 2
        elif self.split:
            return self.width / 2
        else:
            return self.width / (2 * len(self.hue_names))

    def draw_violins(self, ax):
        """Draw the violins onto `ax`."""
        fill_func = ax.fill_betweenx if self.orient == "v" else ax.fill_between
        for i, group_data in enumerate(self.plot_data):

            kws = dict(edgecolor=self.gray, linewidth=self.linewidth)

            # Option 1: we have a single level of grouping
            # -----
            if self.plot_hues is None:

                support, density = self.support[i], self.density[i]

                # Handle special case of no observations in this bin
                if support.size == 0:
                    continue

                # Handle special case of a single observation
                elif support.size == 1:
                    val = support.item()
                    d = density.item()
                    self.draw_single_observation(ax, i, val, d)

```

```

        continue

    # Draw the violin for this group
    grid = np.ones(self.gridsize) * i
    fill_func(support,
              grid - density * self.dwidth,
              grid + density * self.dwidth,
              facecolor=self.colors[i],
              **kws)

    # Draw the interior representation of the data
    if self.inner is None:
        continue

    # Get a nan-free vector of datapoints
    violin_data = remove_na(group_data)

    # Draw box and whisker information
    if self.inner.startswith("box"):
        self.draw_box_lines(ax, violin_data, support, density, i)

    # Draw quartile lines
    elif self.inner.startswith("quart"):
        self.draw_quartiles(ax, violin_data, support, density, i)

    # Draw stick observations
    elif self.inner.startswith("stick"):
        self.draw_stick_lines(ax, violin_data, support, density, i)

    # Draw point observations
    elif self.inner.startswith("point"):
        self.draw_points(ax, violin_data, i)

# Option 2: we have nested grouping by a hue variable
# -----
else:
    offsets = self.hue_offsets
    for j, hue_level in enumerate(self.hue_names):

        support, density = self.support[i][j], self.density[i][j]
        kws["facecolor"] = self.colors[j]

        # Add legend data, but just for one set of violins
        if not i:
            self.add_legend_data(ax, self.colors[j], hue_level)

        # Handle the special case where we have no observations
        if support.size == 0:
            continue

        # Handle the special case where we have one observation
        elif support.size == 1:
            val = support.item()
            d = density.item()
            if self.split:
                d = d / 2
            at_group = i + offsets[j]
            self.draw_single_observation(ax, at_group, val, d)
            continue

# Option 2a: we are drawing a single split violin
# -----

```

```

        if self.split:

            grid = np.ones(self.gridsize) * i
            if j:
                fill_func(support,
                           grid,
                           grid + density * self.dwidth,
                           **kws)
            else:
                fill_func(support,
                           grid - density * self.dwidth,
                           grid,
                           **kws)

        # Draw the interior representation of the data
        if self.inner is None:
            continue

        # Get a nan-free vector of datapoints
        hue_mask = self.plot_hues[i] == hue_level
        violin_data = remove_na(group_data[hue_mask])

        # Draw quartile lines
        if self.inner.startswith("quart"):
            self.draw_quartiles(ax, violin_data,
                                 support, density, i,
                                 ["left", "right"][j])

        # Draw stick observations
        elif self.inner.startswith("stick"):
            self.draw_stick_lines(ax, violin_data,
                                  support, density, i,
                                  ["left", "right"][j])

        # The box and point interior plots are drawn for
        # all data at the group level, so we just do that once
        if not j:
            continue

        # Get the whole vector for this group level
        violin_data = remove_na(group_data)

        # Draw box and whisker information
        if self.inner.startswith("box"):
            self.draw_box_lines(ax, violin_data,
                                support, density, i)

        # Draw point observations
        elif self.inner.startswith("point"):
            self.draw_points(ax, violin_data, i)

        # Option 2b: we are drawing full nested violins
        # -----
    else:
        grid = np.ones(self.gridsize) * (i + offsets[j])
        fill_func(support,
                  grid - density * self.dwidth,
                  grid + density * self.dwidth,
                  **kws)

    # Draw the interior representation

```

```

        if self.inner is None:
            continue

        # Get a nan-free vector of datapoints
        hue_mask = self.plot_hues[i] == hue_level
        violin_data = remove_na(group_data[hue_mask])

        # Draw box and whisker information
        if self.inner.startswith("box"):
            self.draw_box_lines(ax, violin_data,
                                support, density,
                                i + offsets[j])

        # Draw quartile lines
        elif self.inner.startswith("quart"):
            self.draw_quartiles(ax, violin_data,
                                support, density,
                                i + offsets[j])

        # Draw stick observations
        elif self.inner.startswith("stick"):
            self.draw_stick_lines(ax, violin_data,
                                  support, density,
                                  i + offsets[j])

        # Draw point observations
        elif self.inner.startswith("point"):
            self.draw_points(ax, violin_data, i + offsets[j])

def draw_single_observation(self, ax, at_group, at_quant, density):
    """Draw a line to mark a single observation."""
    d_width = density * self.dwidth
    if self.orient == "v":
        ax.plot([at_group - d_width, at_group + d_width],
                [at_quant, at_quant],
                color=self.gray,
                linewidth=self.linewidth)
    else:
        ax.plot([at_quant, at_quant],
                [at_group - d_width, at_group + d_width],
                color=self.gray,
                linewidth=self.linewidth)

def draw_box_lines(self, ax, data, support, density, center):
    """Draw boxplot information at center of the density."""
    # Compute the boxplot statistics
    q25, q50, q75 = np.percentile(data, [25, 50, 75])
    whisker_lim = 1.5 * iqr(data)
    h1 = np.min(data[data >= (q25 - whisker_lim)])
    h2 = np.max(data[data <= (q75 + whisker_lim)])

    # Draw a boxplot using lines and a point
    if self.orient == "v":
        ax.plot([center, center], [h1, h2],
                linewidth=self.linewidth,
                color=self.gray)
        ax.plot([center, center], [q25, q75],
                linewidth=self.linewidth * 3,
                color=self.gray)
        ax.scatter(center, q50,
                  zorder=3,
                  color="white",
                  edgecolor=self.gray,

```

```

                s=np.square(self.linewidth * 2))
else:
    ax.plot([h1, h2], [center, center],
            linewidth=self.linewidth,
            color=self.gray)
    ax.plot([q25, q75], [center, center],
            linewidth=self.linewidth * 3,
            color=self.gray)
    ax.scatter(q50, center,
               zorder=3,
               color="white",
               edgecolor=self.gray,
               s=np.square(self.linewidth * 2))

def draw_quartiles(self, ax, data, support, density, center, split=False):
    """Draw the quartiles as lines at width of density."""
    q25, q50, q75 = np.percentile(data, [25, 50, 75])

    self.draw_to_density(ax, center, q25, support, density, split,
                         linewidth=self.linewidth,
                         dashes=[self.linewidth * 1.5] * 2)
    self.draw_to_density(ax, center, q50, support, density, split,
                         linewidth=self.linewidth,
                         dashes=[self.linewidth * 3] * 2)
    self.draw_to_density(ax, center, q75, support, density, split,
                         linewidth=self.linewidth,
                         dashes=[self.linewidth * 1.5] * 2)

def draw_points(self, ax, data, center):
    """Draw individual observations as points at middle of the violin."""
    kws = dict(s=np.square(self.linewidth * 2),
               color=self.gray,
               edgecolor=self.gray)

    grid = np.ones(len(data)) * center

    if self.orient == "v":
        ax.scatter(grid, data, **kws)
    else:
        ax.scatter(data, grid, **kws)

def draw_stick_lines(self, ax, data, support, density,
                     center, split=False):
    """Draw individual observations as sticks at width of density."""
    for val in data:
        self.draw_to_density(ax, center, val, support, density, split,
                             linewidth=self.linewidth * .5)

def draw_to_density(self, ax, center, val, support, density, split, **kws):
    """Draw a line orthogonal to the value axis at width of density."""
    idx = np.argmin(np.abs(support - val))
    width = self.dwidth * density[idx] * .99

    kws["color"] = self.gray

    if self.orient == "v":
        if split == "left":
            ax.plot([center - width, center], [val, val], **kws)
        elif split == "right":
            ax.plot([center, center + width], [val, val], **kws)
        else:
            ax.plot([center - width, center + width], [val, val], **kws)
    else:

```

```

        if split == "left":
            ax.plot([val, val], [center - width, center], **kws)
        elif split == "right":
            ax.plot([val, val], [center, center + width], **kws)
        else:
            ax.plot([val, val], [center - width, center + width], **kws)

    def plot(self, ax):
        """Make the violin plot."""
        self.draw_violins(ax)
        self.annotate_axes(ax)
        if self.orient == "h":
            ax.invert_yaxis()

class _CategoricalScatterPlotter(_CategoricalPlotter):

    default_palette = "dark"

    @property
    def point_colors(self):
        """Return an index into the palette for each scatter point."""
        point_colors = []
        for i, group_data in enumerate(self.plot_data):

            # Initialize the array for this group level
            group_colors = np.empty(group_data.size, np.int)
            if isinstance(group_data, pd.Series):
                group_colors = pd.Series(group_colors, group_data.index)

            if self.plot_hues is None:

                # Use the same color for all points at this level
                # group_color = self.colors[i]
                group_colors[:] = i

            else:

                # Color the points based on the hue level

                for j, level in enumerate(self.hue_names):
                    # hue_color = self.colors[j]
                    if group_data.size:
                        group_colors[self.plot_hues[i] == level] = j

            point_colors.append(group_colors)

        return point_colors

    def add_legend_data(self, ax):
        """Add empty scatterplot artists with labels for the legend."""
        if self.hue_names is not None:
            for rgb, label in zip(self.colors, self.hue_names):
                ax.scatter([], [],
                           color=mpl.colors.rgb2hex(rgb),
                           label=label,
                           s=60)

class _StripPlotter(_CategoricalScatterPlotter):
    """1-d scatterplot with categorical organization."""
    def __init__(self, x, y, hue, data, order, hue_order,
                 jitter, dodge, orient, color, palette):

```

```

"""Initialize the plotter."""
self._establish_variables(x, y, hue, data, orient, order, hue_order)
self._establish_colors(color, palette, 1)

# Set object attributes
self.dodge = dodge
self.width = .8

if jitter == 1: # Use a good default for `jitter = True`
    jlim = 0.1
else:
    jlim = float(jitter)
if self.hue_names is not None and dodge:
    jlim /= len(self.hue_names)
self.jitterer = stats.uniform(-jlim, jlim * 2).rvs

def draw_stripplot(self, ax, kws):
    """Draw the points onto `ax`."""
    palette = np.asarray(self.colors)
    for i, group_data in enumerate(self.plot_data):
        if self.plot_hues is None or not self.dodge:

            if self.hue_names is None:
                hue_mask = np.ones(group_data.size, np.bool)
            else:
                hue_mask = np.array([h in self.hue_names
                                    for h in self.plot_hues[i]], np.bool)
                # Broken on older numpy's
                # hue_mask = np.in1d(self.plot_hues[i], self.hue_names)

            strip_data = group_data[hue_mask]
            point_colors = np.asarray(self.point_colors[i][hue_mask])

            # Plot the points in centered positions
            cat_pos = np.ones(strip_data.size) * i
            cat_pos += self.jitterer(len(strip_data))
            kws.update(c=palette[point_colors])
            if self.orient == "v":
                ax.scatter(cat_pos, strip_data, **kws)
            else:
                ax.scatter(strip_data, cat_pos, **kws)

        else:
            offsets = self.hue_offsets
            for j, hue_level in enumerate(self.hue_names):
                hue_mask = self.plot_hues[i] == hue_level
                strip_data = group_data[hue_mask]

                point_colors = np.asarray(self.point_colors[i][hue_mask])

                # Plot the points in centered positions
                center = i + offsets[j]
                cat_pos = np.ones(strip_data.size) * center
                cat_pos += self.jitterer(len(strip_data))
                kws.update(c=palette[point_colors])
                if self.orient == "v":
                    ax.scatter(cat_pos, strip_data, **kws)
                else:
                    ax.scatter(strip_data, cat_pos, **kws)

def plot(self, ax, kws):
    """Make the plot."""
    self.draw_stripplot(ax, kws)

```

```

        self.add_legend_data(ax)
        self.annotate_axes(ax)
        if self.orient == "h":
            ax.invert_yaxis()

class _SwarmPlotter(_CategoricalScatterPlotter):

    def __init__(self, x, y, hue, data, order, hue_order,
                 dodge, orient, color, palette):
        """Initialize the plotter."""
        self.establish_variables(x, y, hue, data, orient, order, hue_order)
        self.establish_colors(color, palette, 1)

        # Set object attributes
        self.dodge = dodge
        self.width = .8

    def could_overlap(self, xy_i, swarm, d):
        """Return a list of all swarm points that could overlap with target.

        Assumes that swarm is a sorted list of all points below xy_i.
        """
        _, y_i = xy_i
        neighbors = []
        for xy_j in reversed(swarm):
            _, y_j = xy_j
            if (y_i - y_j) < d:
                neighbors.append(xy_j)
            else:
                break
        return np.array(list(reversed(neighbors)))

    def position_candidates(self, xy_i, neighbors, d):
        """Return a list of (x, y) coordinates that might be valid."""
        candidates = [xy_i]
        x_i, y_i = xy_i
        left_first = True
        for x_j, y_j in neighbors:
            dy = y_i - y_j
            dx = np.sqrt(max(d ** 2 - dy ** 2, 0)) * 1.05
            cl, cr = (x_j - dx, y_i), (x_j + dx, y_i)
            if left_first:
                new_candidates = [cl, cr]
            else:
                new_candidates = [cr, cl]
            candidates.extend(new_candidates)
            left_first = not left_first
        return np.array(candidates)

    def first_non_overlapping_candidate(self, candidates, neighbors, d):
        """Remove candidates from the list if they overlap with the swarm."""

        # IF we have no neighbours, all candidates are good.
        if len(neighbors) == 0:
            return candidates[0]

        neighbors_x = neighbors[:, 0]
        neighbors_y = neighbors[:, 1]

        d_square = d ** 2

        for xy_i in candidates:

```

```

x_i, y_i = xy_i

dx = neighbors_x - x_i
dy = neighbors_y - y_i

sq_distances = np.power(dx, 2.0) + np.power(dy, 2.0)

# good candidate does not overlap any of neighbors
# which means that squared distance between candidate
# and any of the neighbours has to be at least
# square of the diameter
good_candidate = np.all(sq_distances >= d_square)

if good_candidate:
    return xy_i

# If `position_candidates` works well
# this should never happen
raise Exception('No non-overlapping candidates found. '
                 'This should not happen.')

def beeswarm(self, orig_xy, d):
    """Adjust x position of points to avoid overlaps."""
    # In this method, ``x`` is always the categorical axis
    # Center of the swarm, in point coordinates
    midline = orig_xy[0, 0]

    # Start the swarm with the first point
    swarm = [orig_xy[0]]

    # Loop over the remaining points
    for xy_i in orig_xy[1:]:

        # Find the points in the swarm that could possibly
        # overlap with the point we are currently placing
        neighbors = self.could_overlap(xy_i, swarm, d)

        # Find positions that would be valid individually
        # with respect to each of the swarm neighbors
        candidates = self.position_candidates(xy_i, neighbors, d)

        # Sort candidates by their centrality
        offsets = np.abs(candidates[:, 0] - midline)
        candidates = candidates[np.argsort(offsets)]

        # Find the first candidate that does not overlap any neighbours
        new_xy_i = self.first_non_overlapping_candidate(candidates,
                                                       neighbors, d)

        # Place it into the swarm
        swarm.append(new_xy_i)

    return np.array(swarm)

def add_gutters(self, points, center, width):
    """Stop points from extending beyond their territory."""
    half_width = width / 2
    low_gutter = center - half_width
    off_low = points < low_gutter
    if off_low.any():
        points[off_low] = low_gutter
    high_gutter = center + half_width
    off_high = points > high_gutter

```

```

        if off_high.any():
            points[off_high] = high_gutter
        return points

    def swarm_points(self, ax, points, center, width, s, **kws):
        """Find new positions on the categorical axis for each point."""
        # Convert from point size (area) to diameter
        default_lw = mpl.rcParams["patch.linewidth"]
        lw = kws.get("linewidth", kws.get("lw", default_lw))
        dpi = ax.figure.dpi
        d = (np.sqrt(s) + lw) * (dpi / 72)

        # Transform the data coordinates to point coordinates.
        # We'll figure out the swarm positions in the latter
        # and then convert back to data coordinates and replot
        orig_xy = ax.transData.transform(points.get_offsets())

        # Order the variables so that x is the categorical axis
        if self.orient == "h":
            orig_xy = orig_xy[:, [1, 0]]

        # Do the beeswarm in point coordinates
        new_xy = self.beeswarm(orig_xy, d)

        # Transform the point coordinates back to data coordinates
        if self.orient == "h":
            new_xy = new_xy[:, [1, 0]]
        new_x, new_y = ax.transData.inverted().transform(new_xy).T

        # Add gutters
        if self.orient == "v":
            self.add_gutters(new_x, center, width)
        else:
            self.add_gutters(new_y, center, width)

        # Reposition the points so they do not overlap
        points.set_offsets(np.c_[new_x, new_y])

    def draw_swarmplot(self, ax, kws):
        """Plot the data."""
        s = kws.pop("s")

        centers = []
        swarms = []

        palette = np.asarray(self.colors)

        # Set the categorical axes limits here for the swarm math
        if self.orient == "v":
            ax.set_xlim(-.5, len(self.plot_data) - .5)
        else:
            ax.set_ylim(-.5, len(self.plot_data) - .5)

        # Plot each swarm
        for i, group_data in enumerate(self.plot_data):

            if self.plot_hues is None or not self.dodge:

                width = self.width

                if self.hue_names is None:
                    hue_mask = np.ones(group_data.size, np.bool)
                else:

```

```

        hue_mask = np.array([h in self.hue_names
                             for h in self.plot_hues[i]], np.bool)
        # Broken on older numpy's
        # hue_mask = np.in1d(self.plot_hues[i], self.hue_names)

    swarm_data = np.asarray(group_data[hue_mask])
    point_colors = np.asarray(self.point_colors[i][hue_mask])

    # Sort the points for the beeswarm algorithm
    sorter = np.argsort(swarm_data)
    swarm_data = swarm_data[sorter]
    point_colors = point_colors[sorter]

    # Plot the points in centered positions
    cat_pos = np.ones(swarm_data.size) * i
    kws.update(c=palette[point_colors])
    if self.orient == "v":
        points = ax.scatter(cat_pos, swarm_data, s=s, **kws)
    else:
        points = ax.scatter(swarm_data, cat_pos, s=s, **kws)

    centers.append(i)
    swarms.append(points)

else:
    offsets = self.hue_offsets
    width = self.nested_width

    for j, hue_level in enumerate(self.hue_names):
        hue_mask = self.plot_hues[i] == hue_level
        swarm_data = np.asarray(group_data[hue_mask])
        point_colors = np.asarray(self.point_colors[i][hue_mask])

        # Sort the points for the beeswarm algorithm
        sorter = np.argsort(swarm_data)
        swarm_data = swarm_data[sorter]
        point_colors = point_colors[sorter]

        # Plot the points in centered positions
        center = i + offsets[j]
        cat_pos = np.ones(swarm_data.size) * center
        kws.update(c=palette[point_colors])
        if self.orient == "v":
            points = ax.scatter(cat_pos, swarm_data, s=s, **kws)
        else:
            points = ax.scatter(swarm_data, cat_pos, s=s, **kws)

        centers.append(center)
        swarms.append(points)

# Autoscale the values axis to set the data/axes transforms properly
ax.autoscale_view(scalex=self.orient == "h", scaley=self.orient == "v")

# Update the position of each point on the categorical axis
# Do this after plotting so that the numerical axis limits are correct
for center, swarm in zip(centers, swarms):
    if swarm.get_offsets().size:
        self.swarm_points(ax, swarm, center, width, s, **kws)

def plot(self, ax, kws):
    """Make the full plot."""
    self.draw_swarmplot(ax, kws)
    self.add_legend_data(ax)

```

```

        self.annotate_axes(ax)
        if self.orient == "h":
            ax.invert_yaxis()

class _CategoricalStatPlotter(_CategoricalPlotter):

    @property
    def nested_width(self):
        """A float with the width of plot elements when hue nesting is used."""
        if self.dodge:
            width = self.width / len(self.hue_names)
        else:
            width = self.width
        return width

    def estimate_statistic(self, estimator, ci, n_boot, seed):

        if self.hue_names is None:
            statistic = []
            confint = []
        else:
            statistic = [[] for _ in self.plot_data]
            confint = [[] for _ in self.plot_data]

        for i, group_data in enumerate(self.plot_data):

            # Option 1: we have a single layer of grouping
            # -----
            if self.plot_hues is None:

                if self.plot_units is None:
                    stat_data = remove_na(group_data)
                    unit_data = None
                else:
                    unit_data = self.plot_units[i]
                    have = pd.notnull(np.c_[group_data, unit_data]).all(axis=1)
                    stat_data = group_data[have]
                    unit_data = unit_data[have]

                # Estimate a statistic from the vector of data
                if not stat_data.size:
                    statistic.append(np.nan)
                else:
                    statistic.append(estimator(stat_data))

            # Get a confidence interval for this estimate
            if ci is not None:

                if stat_data.size < 2:
                    confint.append([np.nan, np.nan])
                    continue

                if ci == "sd":

                    estimate = estimator(stat_data)
                    sd = np.std(stat_data)
                    confint.append((estimate - sd, estimate + sd))

                else:

                    boots = bootstrap(stat_data, func=estimator,

```

```

        n_boot=n_boot,
        units=unit_data,
        seed=seed)
confint.append(utils.ci(boots, ci))

# Option 2: we are grouping by a hue layer
# -----
else:
    for j, hue_level in enumerate(self.hue_names):

        if not self.plot_hues[i].size:
            statistic[i].append(np.nan)
            if ci is not None:
                confint[i].append((np.nan, np.nan))
            continue

        hue_mask = self.plot_hues[i] == hue_level
        if self.plot_units is None:
            stat_data = remove_na(group_data[hue_mask])
            unit_data = None
        else:
            group_units = self.plot_units[i]
            have = pd.notnull(
                np.c_[group_data, group_units]
            ).all(axis=1)
            stat_data = group_data[hue_mask & have]
            unit_data = group_units[hue_mask & have]

        # Estimate a statistic from the vector of data
        if not stat_data.size:
            statistic[i].append(np.nan)
        else:
            statistic[i].append(estimator(stat_data))

        # Get a confidence interval for this estimate
        if ci is not None:

            if stat_data.size < 2:
                confint[i].append([np.nan, np.nan])
                continue

            if ci == "sd":

                estimate = estimator(stat_data)
                sd = np.std(stat_data)
                confint[i].append((estimate - sd, estimate + sd))

            else:

                boots = bootstrap(stat_data, func=estimator,
                                   n_boot=n_boot,
                                   units=unit_data,
                                   seed=seed)
                confint[i].append(utils.ci(boots, ci))

        # Save the resulting values for plotting
        self.statistic = np.array(statistic)
        self.confint = np.array(confint)

def draw_confints(self, ax, at_group, confint, colors,
                  errwidth=None, capsizer=None, **kws):

```

```

if errwidth is not None:
    kws.setdefault("lw", errwidth)
else:
    kws.setdefault("lw", mpl.rcParams["lines.linewidth"] * 1.8)

for at, (ci_low, ci_high), color in zip(at_group,
                                         confint,
                                         colors):
    if self.orient == "v":
        ax.plot([at, at], [ci_low, ci_high], color=color, **kws)
        if capsizes is not None:
            ax.plot([at - capsizes / 2, at + capsizes / 2],
                    [ci_low, ci_low], color=color, **kws)
            ax.plot([at - capsizes / 2, at + capsizes / 2],
                    [ci_high, ci_high], color=color, **kws)
    else:
        ax.plot([ci_low, ci_high], [at, at], color=color, **kws)
        if capsizes is not None:
            ax.plot([ci_low, ci_low],
                    [at - capsizes / 2, at + capsizes / 2],
                    color=color, **kws)
            ax.plot([ci_high, ci_high],
                    [at - capsizes / 2, at + capsizes / 2],
                    color=color, **kws)

class _BarPlotter(_CategoricalStatPlotter):
    """Show point estimates and confidence intervals with bars."""

    def __init__(self, x, y, hue, data, order, hue_order,
                 estimator, ci, n_boot, units, seed,
                 orient, color, palette, saturation, errcolor,
                 errwidth, capsizes, dodge):
        """Initialize the plotter."""
        self.establish_variables(x, y, hue, data, orient,
                               order, hue_order, units)
        self.establish_colors(color, palette, saturation)
        self.estimate_statistic(estimator, ci, n_boot, seed)

        self.dodge = dodge

        self.errcolor = errcolor
        self.errwidth = errwidth
        self.capsizes = capsizes

    def draw_bars(self, ax, kws):
        """Draw the bars onto `ax`."""
        # Get the right matplotlib function depending on the orientation
        barfunc = ax.bar if self.orient == "v" else ax.bars
        barpos = np.arange(len(self.statistic))

        if self.plot_hues is None:
            # Draw the bars
            barfunc(barpos, self.statistic, self.width,
                    color=self.colors, align="center", **kws)

            # Draw the confidence intervals
            errcolors = [self.errcolor] * len(barpos)
            self.draw_confints(ax,
                               barpos,
                               self.confint,
                               errcolors,

```

```

        self.errwidth,
        self.capsize)

else:

    for j, hue_level in enumerate(self.hue_names):

        # Draw the bars
        offpos = barpos + self.hue_offsets[j]
        barfunc(offpos, self.statistic[:, j], self.nested_width,
                color=self.colors[j], align="center",
                label=hue_level, **kws)

        # Draw the confidence intervals
        if self.confint.size:
            confint = self.confint[:, j]
            errcolors = [self.errcolor] * len(offpos)
            self.draw_confints(ax,
                               offpos,
                               confint,
                               errcolors,
                               self.errwidth,
                               self.capsize)

def plot(self, ax, bar_kws):
    """Make the plot."""
    self.draw_bars(ax, bar_kws)
    self.annotate_axes(ax)
    if self.orient == "h":
        ax.invert_yaxis()

class _PointPlotter(_CategoricalStatPlotter):

    default_palette = "dark"

    """Show point estimates and confidence intervals with (joined) points."""
    def __init__(self, x, y, hue, data, order, hue_order,
                 estimator, ci, n_boot, units, seed,
                 markers, linestyles, dodge, join, scale,
                 orient, color, palette, errwidth=None, capsize=None):
        """Initialize the plotter."""
        self.establish_variables(x, y, hue, data, orient,
                                order, hue_order, units)
        self.establish_colors(color, palette, 1)
        self.estimate_statistic(estimator, ci, n_boot, seed)

        # Override the default palette for single-color plots
        if hue is None and color is None and palette is None:
            self.colors = [color_palette()[0]] * len(self.colors)

        # Don't join single-layer plots with different colors
        if hue is None and palette is not None:
            join = False

        # Use a good default for `dodge=True`
        if dodge is True and self.hue_names is not None:
            dodge = .025 * len(self.hue_names)

        # Make sure we have a marker for each hue level
        if isinstance(markers, str):
            markers = [markers] * len(self.colors)
        self.markers = markers

```

```

# Make sure we have a line style for each hue level
if isinstance(linestyles, str):
    linestyles = [linestyles] * len(self.colors)
self.linestyles = linestyles

# Set the other plot components
self.dodge = dodge
self.join = join
self.scale = scale
self.errwidth = errwidth
self.capsize = capsiz

@property
def hue_offsets(self):
    """Offsets relative to the center position for each hue level."""
    if self.dodge:
        offset = np.linspace(0, self.dodge, len(self.hue_names))
        offset -= offset.mean()
    else:
        offset = np.zeros(len(self.hue_names))
    return offset

def draw_points(self, ax):
    """Draw the main data components of the plot."""
    # Get the center positions on the categorical axis
    pointpos = np.arange(len(self.statistic))

    # Get the size of the plot elements
    lw = mpl.rcParams["lines.linewidth"] * 1.8 * self.scale
    mew = lw * .75
    markersize = np.pi * np.square(lw) * 2

    if self.plot_hues is None:

        # Draw lines joining each estimate point
        if self.join:
            color = self.colors[0]
            ls = self.linestyles[0]
            if self.orient == "h":
                ax.plot(self.statistic, pointpos,
                        color=color, ls=ls, lw=lw)
            else:
                ax.plot(pointpos, self.statistic,
                        color=color, ls=ls, lw=lw)

        # Draw the confidence intervals
        self.draw_confints(ax, pointpos, self.confint, self.colors,
                           self.errwidth, self.capsize)

        # Draw the estimate points
        marker = self.markers[0]
        colors = [mpl.colors.colorConverter.to_rgb(c) for c in self.colors]
        if self.orient == "h":
            x, y = self.statistic, pointpos
        else:
            x, y = pointpos, self.statistic
        ax.scatter(x, y,
                   linewidth=mew, marker=marker, s=markersize,
                   facecolor=colors, edgecolor=colors)

    else:

```

```

offsets = self.hue_offsets
for j, hue_level in enumerate(self.hue_names):

    # Determine the values to plot for this level
    statistic = self.statistic[:, j]

    # Determine the position on the categorical and z axes
    offpos = pointpos + offsets[j]
    z = j + 1

    # Draw lines joining each estimate point
    if self.join:
        color = self.colors[j]
        ls = self.linestyles[j]
        if self.orient == "h":
            ax.plot(statistic, offpos, color=color,
                    zorder=z, ls=ls, lw=lw)
        else:
            ax.plot(offpos, statistic, color=color,
                    zorder=z, ls=ls, lw=lw)

    # Draw the confidence intervals
    if self.confint.size:
        confint = self.confint[:, j]
        errcolors = [self.colors[j]] * len(offpos)
        self.draw_confints(ax, offpos, confint, errcolors,
                           self.errwidth, self.capsize,
                           zorder=z)

    # Draw the estimate points
    n_points = len(remove_na(offpos))
    marker = self.markers[j]
    color = mpl.colors.colorConverter.to_rgb(self.colors[j])

    if self.orient == "h":
        x, y = statistic, offpos
    else:
        x, y = offpos, statistic

    if not len(remove_na(statistic)):
        x = y = [np.nan] * n_points

    ax.scatter(x, y, label=hue_level,
               facecolor=color, edgecolor=color,
               linewidth=mew, marker=marker, s=markersize,
               zorder=z)

def plot(self, ax):
    """Make the plot."""
    self.draw_points(ax)
    self.annotate_axes(ax)
    if self.orient == "h":
        ax.invert_yaxis()

class _LVPlotter(_CategoricalPlotter):

    def __init__(self, x, y, hue, data, order, hue_order,
                 orient, color, palette, saturation,
                 width, dodge, k_depth, linewidth, scale, outlier_prop,
                 showfliers=True):

        # TODO assigning variables for None is unneccesary

```

```

if width is None:
    width = .8
self.width = width

self.dodge = dodge

if saturation is None:
    saturation = .75
self.saturation = saturation

if k_depth is None:
    k_depth = 'proportion'
self.k_depth = k_depth

if linewidth is None:
    linewidth = mpl.rcParams["lines.linewidth"]
self.linewidth = linewidth

if scale is None:
    scale = 'exponential'
self.scale = scale

self.outlier_prop = outlier_prop

self.showfliers = showfliers

self.establish_variables(x, y, hue, data, orient, order, hue_order)
self.establish_colors(color, palette, saturation)

def lv_box_ends(self, vals, k_depth='proportion', outlier_prop=None):
    """Get the number of data points and calculate `depth` of
    letter-value plot."""
    vals = np.asarray(vals)
    vals = vals[np.isfinite(vals)]
    n = len(vals)
    # If p is not set, calculate it so that 8 points are outliers
    if not outlier_prop:
        # Conventional boxplots assume this proportion of the data are
        # outliers.
        p = 0.007
    else:
        if ((outlier_prop > 1.) or (outlier_prop < 0.)):
            raise ValueError('outlier_prop not in range [0, 1]!')
        p = outlier_prop
    # Select the depth, i.e. number of boxes to draw, based on the method
    k_dict = {'proportion': (np.log2(n)) - int(np.log2(n*p)) + 1,
              'tukey': (np.log2(n)) - 3,
              'trustworthy': (np.log2(n) -
                               np.log2(2*stats.norm.ppf((1-p)**2))) + 1}
    k = k_dict[k_depth]
    try:
        k = int(k)
    except ValueError:
        k = 1
    # If the number happens to be less than 0, set k to 0
    if k < 1.:
        k = 1
    # Calculate the upper box ends
    upper = [100*(1 - 0.5**i) for i in range(k, -1, -1)]
    # Calculate the lower box ends
    lower = [100*(0.5**i) for i in range(k, -1, -1)]
    # Stitch the box ends together
    percentile_ends = [(i, j) for i, j in zip(lower, upper)]

```



```

# Scale the width of the boxes so the biggest starts at 1
w_area = np.array([width(height(b), i, k)
                  for i, b in enumerate(box_ends)])
w_area = w_area / np.max(w_area)

# Calculate the medians
y = np.median(box_data)

# Calculate the outliers and plot (only if showfliers == True)
outliers = []
if self.showfliers:
    outliers = self._lv_outliers(box_data, k)
hex_color = mpl.colors.rgb2hex(color)

if vert:
    boxes = [vert_perc_box(x, b[0], i, k, b[1])
              for i, b in enumerate(zip(box_ends, w_area))]

    # Plot the medians
    ax.plot([x - widths / 2, x + widths / 2], [y, y],
            c='15', alpha=.45, **kws)

    ax.scatter(np.repeat(x, len(outliers)), outliers,
               marker='d', c=hex_color, **kws)
else:
    boxes = [horz_perc_box(x, b[0], i, k, b[1])
              for i, b in enumerate(zip(box_ends, w_area))]

    # Plot the medians
    ax.plot([y, y], [x - widths / 2, x + widths / 2],
            c='15', alpha=.45, **kws)

    ax.scatter(outliers, np.repeat(x, len(outliers)),
               marker='d', c=hex_color, **kws)

# Construct a color map from the input color
rgb = [[1, 1, 1], hex_color]
cmap = mpl.colors.LinearSegmentedColormap.from_list('new_map', rgb)
collection = PatchCollection(boxes, cmap=cmap)

# Set the color gradation
collection.set_array(np.array(np.linspace(0, 1, len(boxes)))) 

# Plot the boxes
ax.add_collection(collection)

def draw_letter_value_plot(self, ax, kws):
    """Use matplotlib to draw a letter value plot on an Axes."""
    vert = self.orient == "v"

    for i, group_data in enumerate(self.plot_data):

        if self.plot_hues is None:

            # Handle case where there is data at this level
            if group_data.size == 0:
                continue

            # Draw a single box or a set of boxes
            # with a single level of grouping
            box_data = remove_na(group_data)

```

```

        # Handle case where there is no non-null data
        if box_data.size == 0:
            continue

        color = self.colors[i]

        self._lvplot(box_data,
                     positions=[i],
                     color=color,
                     vert=vert,
                     widths=self.width,
                     k_depth=self.k_depth,
                     ax=ax,
                     scale=self.scale,
                     outlier_prop=self.outlier_prop,
                     showfliers=self.showfliers,
                     **kws)

    else:
        # Draw nested groups of boxes
        offsets = self.hue_offsets
        for j, hue_level in enumerate(self.hue_names):

            # Add a legend for this hue level
            if not i:
                self.add_legend_data(ax, self.colors[j], hue_level)

            # Handle case where there is data at this level
            if group_data.size == 0:
                continue

            hue_mask = self.plot_hues[i] == hue_level
            box_data = remove_na(group_data[hue_mask])

            # Handle case where there is no non-null data
            if box_data.size == 0:
                continue

            color = self.colors[j]
            center = i + offsets[j]
            self._lvplot(box_data,
                         positions=[center],
                         color=color,
                         vert=vert,
                         widths=self.nested_width,
                         k_depth=self.k_depth,
                         ax=ax,
                         scale=self.scale,
                         outlier_prop=self.outlier_prop,
                         **kws)

def plot(self, ax, boxplot_kws):
    """Make the plot."""
    self.draw_letter_value_plot(ax, boxplot_kws)
    self.annotate_axes(ax)
    if self.orient == "h":
        ax.invert_yaxis()

_categorical_docs = dict(
    # Shared narrative docs
    categorical_narrative=dedent("""\

```

This function always treats one of the variables as categorical and draws data at ordinal positions (0, 1, ... n) on the relevant axis, even when the data has a numeric or date type.

See the :ref:`tutorial <categorical_tutorial>` for more information.\n"""),
main_api_narrative=dedent("""\

Input data can be passed in a variety of formats, including:

- Vectors of data represented as lists, numpy arrays, or pandas Series objects passed directly to the ``x``, ``y``, and/or ``hue`` parameters.
- A "long-form" DataFrame, in which case the ``x``, ``y``, and ``hue`` variables will determine how the data are plotted.
- A "wide-form" DataFrame, such that each numeric column will be plotted.
- An array or list of vectors.

In most cases, it is possible to use numpy or Python objects, but pandas objects are preferable because the associated names will be used to annotate the axes. Additionally, you can use Categorical types for the grouping variables to control the order of plot elements.\n"""),

```
# Shared function parameters
input_params=dedent("""\
x, y, hue : names of variables in ``data`` or vector data, optional
    Inputs for plotting long-form data. See examples for interpretation.\n"""
),
string_input_params=dedent("""\
x, y, hue : names of variables in ``data``
    Inputs for plotting long-form data. See examples for interpretation.\n"""
),
categorical_data=dedent("""\
data : DataFrame, array, or list of arrays, optional
    Dataset for plotting. If ``x`` and ``y`` are absent, this is
    interpreted as wide-form. Otherwise it is expected to be long-form.\n"""
),
long_form_data=dedent("""\
data : DataFrame
    Long-form (tidy) dataset for plotting. Each column should correspond
    to a variable, and each row should correspond to an observation.\n"""
),
order_vars=dedent("""\
order, hue_order : lists of strings, optional
    Order to plot the categorical levels in, otherwise the levels are
    inferred from the data objects.\n"""
),
stat_api_params=dedent("""\
estimator : callable that maps vector -> scalar, optional
    Statistical function to estimate within each categorical bin.
ci : float or "sd" or None, optional
    Size of confidence intervals to draw around estimated values. If
    "sd", skip bootstrapping and draw the standard deviation of the
    observations. If ``None``, no bootstrapping will be performed, and
    error bars will not be drawn.
n_boot : int, optional
    Number of bootstrap iterations to use when computing confidence
    intervals.
units : name of variable in ``data`` or vector data, optional
    Identifier of sampling units, which will be used to perform a
    multilevel bootstrap and account for repeated measures design.
seed : int, numpy.random.Generator, or numpy.random.RandomState, optional
    Seed or random number generator for reproducible bootstrapping.\
```

```
""",  
orient=dedent("""\  
orient : "v" | "h", optional  
    Orientation of the plot (vertical or horizontal). This is usually  
    inferred from the dtype of the input variables, but can be used to  
    specify when the "categorical" variable is a numeric or when plotting  
    wide-form data.\  
"""),  
color=dedent("""\  
color : matplotlib color, optional  
    Color for all of the elements, or seed for a gradient palette.\  
"""),  
palette=dedent("""\  
palette : palette name, list, or dict, optional  
    Color palette that maps either the grouping variable or the hue  
    variable. If the palette is a dictionary, keys should be names of  
    levels and values should be matplotlib colors.\  
"""),  
saturation=dedent("""\  
saturation : float, optional  
    Proportion of the original saturation to draw colors at. Large patches  
    often look better with slightly desaturated colors, but set this to  
    ``1`` if you want the plot colors to perfectly match the input color  
    spec.\  
"""),  
capsize=dedent("""\  
capsize : float, optional  
    Width of the "caps" on error bars.  
"""),  
errwidth=dedent("""\  
errwidth : float, optional  
    Thickness of error bar lines (and caps).\  
"""),  
width=dedent("""\  
width : float, optional  
    Width of a full element when not using hue nesting, or width of all the  
    elements for one level of the major grouping variable.\  
"""),  
dodge=dedent("""\  
dodge : bool, optional  
    When hue nesting is used, whether elements should be shifted along the  
    categorical axis.\  
"""),  
linewidth=dedent("""\  
linewidth : float, optional  
    Width of the gray lines that frame the plot elements.\  
"""),  
ax_in=dedent("""\  
ax : matplotlib Axes, optional  
    Axes object to draw the plot onto, otherwise uses the current Axes.\  
"""),  
ax_out=dedent("""\  
ax : matplotlib Axes  
    Returns the Axes object with the plot drawn onto it.\  
"""),  
  
# Shared see also  
boxplot=dedent("""\  
boxplot : A traditional box-and-whisker plot with a similar API.\  
"""),  
violinplot=dedent("""\  
violinplot : A combination of boxplot and kernel density estimation.\  
"""),
```

```

stripplot=dedent("""\
stripplot : A scatterplot where one variable is categorical. Can be used
            in conjunction with other plots to show each observation.\"
"""),
swarmplot=dedent("""\
swarmplot : A categorical scatterplot where the points do not overlap. Can
            be used with other plots to show each observation.\"
"""),
barplot=dedent("""\
barplot : Show point estimates and confidence intervals using bars.\"
"""),
countplot=dedent("""\
countplot : Show the counts of observations in each categorical bin.\"
"""),
pointplot=dedent("""\
pointplot : Show point estimates and confidence intervals using scatterplot
            glyphs.\"
"""),
catplot=dedent("""\
catplot : Combine a categorical plot with a :class:`FacetGrid`.\"
"""),
boxenplot=dedent("""\
boxenplot : An enhanced boxplot for larger datasets.\"
"""),
)

```

`_categorical_docs.update(_facet_docs)`

```

def boxplot(x=None, y=None, hue=None, data=None, order=None, hue_order=None,
            orient=None, color=None, palette=None, saturation=.75,
            width=.8, dodge=True, fliersize=5, linewidth=None,
            whis=1.5, ax=None, **kwargs):
    plotter = _BoxPlotter(x, y, hue, data, order, hue_order,
                          orient, color, palette, saturation,
                          width, dodge, fliersize, linewidth)

    if ax is None:
        ax = plt.gca()
    kwargs.update(dict(whis=whis))

    plotter.plot(ax, kwargs)
    return ax

```

`boxplot.__doc__ = dedent("""\
Draw a box plot to show distributions with respect to categories.

A box plot (or box-and-whisker plot) shows the distribution of quantitative
data in a way that facilitates comparisons between variables or across
levels of a categorical variable. The box shows the quartiles of the
dataset while the whiskers extend to show the rest of the distribution,
except for points that are determined to be "outliers" using a method
that is a function of the inter-quartile range.

{main_api_narrative}

{categorical_narrative}

Parameters
-----`

```
{input_params}
{categorical_data}
{order_vars}
{orient}
{color}
{palette}
{saturation}
{width}
{dodge}
fliersize : float, optional
    Size of the markers used to indicate outlier observations.
{linewidth}
whis : float, optional
    Proportion of the IQR past the low and high quartiles to extend the
    plot whiskers. Points outside this range will be identified as
    outliers.
{ax_in}
kwargs : key, value mappings
    Other keyword arguments are passed through to
    :meth:`matplotlib.axes.Axes.boxplot`.
```

Returns

```
{ax_out}
```

See Also

```
{violinplot}
{stripplot}
{swarmplot}
{catplot}
```

Examples

Draw a single horizontal boxplot:

```
.. plot::
   :context: close-figs

>>> import seaborn as sns
>>> sns.set(style="whitegrid")
>>> tips = sns.load_dataset("tips")
>>> ax = sns.boxplot(x=tips["total_bill"])
```

Draw a vertical boxplot grouped by a categorical variable:

```
.. plot::
   :context: close-figs

>>> ax = sns.boxplot(x="day", y="total_bill", data=tips)
```

Draw a boxplot with nested grouping by two categorical variables:

```
.. plot::
   :context: close-figs

>>> ax = sns.boxplot(x="day", y="total_bill", hue="smoker",
...                     data=tips, palette="Set3")
```

Draw a boxplot with nested grouping when some bins are empty:

```
.. plot::
```

```

:context: close-figs

>>> ax = sns.boxplot(x="day", y="total_bill", hue="time",
...                      data=tips, linewidth=2.5)

Control box order by passing an explicit order:

.. plot::
   :context: close-figs

>>> ax = sns.boxplot(x="time", y="tip", data=tips,
...                      order=["Dinner", "Lunch"])

Draw a boxplot for each numeric variable in a DataFrame:

.. plot::
   :context: close-figs

>>> iris = sns.load_dataset("iris")
>>> ax = sns.boxplot(data=iris, orient="h", palette="Set2")

Use ``hue`` without changing box position or width:

.. plot::
   :context: close-figs

>>> tips["weekend"] = tips["day"].isin(["Sat", "Sun"])
>>> ax = sns.boxplot(x="day", y="total_bill", hue="weekend",
...                      data=tips, dodge=False)

Use :func:`swarmplot` to show the datapoints on top of the boxes:

.. plot::
   :context: close-figs

>>> ax = sns.boxplot(x="day", y="total_bill", data=tips)
>>> ax = sns.swarmplot(x="day", y="total_bill", data=tips, color=".25")

Use :func:`catplot` to combine a :func:`boxplot` and a
:class:`FacetGrid`. This allows grouping within additional categorical
variables. Using :func:`catplot` is safer than using :class:`FacetGrid`
directly, as it ensures synchronization of variable order across facets:

.. plot::
   :context: close-figs

>>> g = sns.catplot(x="sex", y="total_bill",
...                     hue="smoker", col="time",
...                     data=tips, kind="box",
...                     height=4, aspect=.7);

""").format(**_categorical_docs)

def violinplot(x=None, y=None, hue=None, data=None, order=None, hue_order=None,
               bw="scott", cut=2, scale="area", scale_hue=True, gridsize=100,
               width=.8, inner="box", split=False, dodge=True, orient=None,
               linewidth=None, color=None, palette=None, saturation=.75,
               ax=None, **kwargs):

    plotter = _ViolinPlotter(x, y, hue, data, order, hue_order,
                           bw, cut, scale, scale_hue, gridsize,
                           width, inner, split, dodge, orient, linewidth,

```

```

        color, palette, saturation)

if ax is None:
    ax = plt.gca()

plotter.plot(ax)
return ax

violinplot.__doc__ = dedent("""\
Draw a combination of boxplot and kernel density estimate.

A violin plot plays a similar role as a box and whisker plot. It shows the
distribution of quantitative data across several levels of one (or more)
categorical variables such that those distributions can be compared. Unlike
a box plot, in which all of the plot components correspond to actual
datapoints, the violin plot features a kernel density estimation of the
underlying distribution.

This can be an effective and attractive way to show multiple distributions
of data at once, but keep in mind that the estimation procedure is
influenced by the sample size, and violins for relatively small samples
might look misleadingly smooth.

{main_api_narrative}

{categorical_narrative}

Parameters
-----
{input_params}
{categorical_data}
{order_vars}
bw : {{'scott', 'silverman', float}}, optional
    Either the name of a reference rule or the scale factor to use when
    computing the kernel bandwidth. The actual kernel size will be
    determined by multiplying the scale factor by the standard deviation of
    the data within each bin.
cut : float, optional
    Distance, in units of bandwidth size, to extend the density past the
    extreme datapoints. Set to 0 to limit the violin range within the range
    of the observed data (i.e., to have the same effect as ``trim=True`` in
    ``ggplot``).
scale : {{"area", "count", "width"}}, optional
    The method used to scale the width of each violin. If ``area``, each
    violin will have the same area. If ``count``, the width of the violins
    will be scaled by the number of observations in that bin. If ``width``,
    each violin will have the same width.
scale_hue : bool, optional
    When nesting violins using a ``hue`` variable, this parameter
    determines whether the scaling is computed within each level of the
    major grouping variable (``scale_hue=True``) or across all the violins
    on the plot (``scale_hue=False``).
gridsize : int, optional
    Number of points in the discrete grid used to compute the kernel
    density estimate.
{width}
inner : {{"box", "quartile", "point", "stick", None}}, optional
    Representation of the datapoints in the violin interior. If ``box``,
    draw a miniature boxplot. If ``quartiles``, draw the quartiles of the
    distribution. If ``point`` or ``stick``, show each underlying
    datapoint. Using ``None`` will draw unadorned violins.
split : bool, optional

```

When using hue nesting with a variable that takes two levels, setting ``split`` to True will draw half of a violin for each level. This can make it easier to directly compare the distributions.

{dodge}
{orient}
{linewidth}
{color}
{palette}
{saturation}
{ax_in}

Returns

{ax_out}

See Also

{boxplot}
{stripplot}
{swarmplot}
{catplot}

Examples

Draw a single horizontal violinplot:

```
.. plot::  
    :context: close-figs  
  
    >>> import seaborn as sns  
    >>> sns.set(style="whitegrid")  
    >>> tips = sns.load_dataset("tips")  
    >>> ax = sns.violinplot(x=tips["total_bill"])
```

Draw a vertical violinplot grouped by a categorical variable:

```
.. plot::  
    :context: close-figs  
  
    >>> ax = sns.violinplot(x="day", y="total_bill", data=tips)
```

Draw a violinplot with nested grouping by two categorical variables:

```
.. plot::  
    :context: close-figs  
  
    >>> ax = sns.violinplot(x="day", y="total_bill", hue="smoker",  
    ...                         data=tips, palette="muted")
```

Draw split violins to compare the across the hue variable:

```
.. plot::  
    :context: close-figs  
  
    >>> ax = sns.violinplot(x="day", y="total_bill", hue="smoker",  
    ...                         data=tips, palette="muted", split=True)
```

Control violin order by passing an explicit order:

```
.. plot::  
    :context: close-figs
```

```
>>> ax = sns.violinplot(x="time", y="tip", data=tips,
...                         order=["Dinner", "Lunch"])
```

Scale the violin width by the number of observations in each bin:

```
.. plot::
   :context: close-figs

>>> ax = sns.violinplot(x="day", y="total_bill", hue="sex",
...                         data=tips, palette="Set2", split=True,
...                         scale="count")
```

Draw the quartiles as horizontal lines instead of a mini-box:

```
.. plot::
   :context: close-figs

>>> ax = sns.violinplot(x="day", y="total_bill", hue="sex",
...                         data=tips, palette="Set2", split=True,
...                         scale="count", inner="quartile")
```

Show each observation with a stick inside the violin:

```
.. plot::
   :context: close-figs

>>> ax = sns.violinplot(x="day", y="total_bill", hue="sex",
...                         data=tips, palette="Set2", split=True,
...                         scale="count", inner="stick")
```

Scale the density relative to the counts across all bins:

```
.. plot::
   :context: close-figs

>>> ax = sns.violinplot(x="day", y="total_bill", hue="sex",
...                         data=tips, palette="Set2", split=True,
...                         scale="count", inner="stick", scale_hue=False)
```

Use a narrow bandwidth to reduce the amount of smoothing:

```
.. plot::
   :context: close-figs

>>> ax = sns.violinplot(x="day", y="total_bill", hue="sex",
...                         data=tips, palette="Set2", split=True,
...                         scale="count", inner="stick",
...                         scale_hue=False, bw=.2)
```

Draw horizontal violins:

```
.. plot::
   :context: close-figs

>>> planets = sns.load_dataset("planets")
>>> ax = sns.violinplot(x="orbital_period", y="method",
...                         data=planets[planets.orbital_period < 1000],
...                         scale="width", palette="Set3")
```

Don't let density extend past extreme values in the data:

```
.. plot::
   :context: close-figs
```

```

>>> ax = sns.violinplot(x="orbital_period", y="method",
...                      data=planets[planets.orbital_period < 1000],
...                      cut=0, scale="width", palette="Set3")

Use ``hue`` without changing violin position or width:

.. plot::
   :context: close-figs

>>> tips["weekend"] = tips["day"].isin(["Sat", "Sun"])
>>> ax = sns.violinplot(x="day", y="total_bill", hue="weekend",
...                       data=tips, dodge=False)

Use :func:`catplot` to combine a :func:`violinplot` and a
:class:`FacetGrid`. This allows grouping within additional categorical
variables. Using :func:`catplot` is safer than using :class:`FacetGrid`
directly, as it ensures synchronization of variable order across facets:

.. plot::
   :context: close-figs

>>> g = sns.catplot(x="sex", y="total_bill",
...                    hue="smoker", col="time",
...                    data=tips, kind="violin", split=True,
...                    height=4, aspect=.7);

""").format(**_categorical_docs)

def lvplot(*args, **kwargs):
    """Deprecated; please use `boxenplot`."""
    msg = (
        "The `lvplot` function has been renamed to `boxenplot`. The original "
        "name will be removed in a future release. Please update your code. "
    )
    warnings.warn(msg)

    return boxenplot(*args, **kwargs)

def boxenplot(x=None, y=None, hue=None, data=None, order=None, hue_order=None,
             orient=None, color=None, palette=None, saturation=.75,
             width=.8, dodge=True, k_depth='proportion', linewidth=None,
             scale='exponential', outlier_prop=None, showfliers=True, ax=None,
             **kwargs):
    plotter = _LVPplotter(x, y, hue, data, order, hue_order,
                          orient, color, palette, saturation,
                          width, dodge, k_depth, linewidth, scale,
                          outlier_prop, showfliers)

    if ax is None:
        ax = plt.gca()

    plotter.plot(ax, kwargs)
    return ax

boxenplot.__doc__ = dedent("""\
    Draw an enhanced box plot for larger datasets.

```

This style of plot was originally named a "letter value" plot because it shows a large number of quantiles that are defined as "letter values". It is similar to a box plot in plotting a nonparametric representation of a distribution in which all features correspond to actual observations. By plotting more quantiles, it provides more information about the shape of the distribution, particularly in the tails. For a more extensive explanation, you can read the paper that introduced the plot:

```
https://vita.had.co.nz/papers/letter-value-plot.html

{main_api_narrative}

{categorical_narrative}

Parameters
-----
{input_params}
{categorical_data}
{order_vars}
{orient}
{color}
{palette}
{saturation}
{width}
{dodge}
k_depth : "proportion" | "tukey" | "trustworthy", optional
    The number of boxes, and by extension number of percentiles, to draw.
    All methods are detailed in Wickham's paper. Each makes different
    assumptions about the number of outliers and leverages different
    statistical properties.
{linewidth}
scale : "linear" | "exponential" | "area"
    Method to use for the width of the letter value boxes. All give similar
    results visually. "linear" reduces the width by a constant linear
    factor, "exponential" uses the proportion of data not covered, "area"
    is proportional to the percentage of data covered.
outlier_prop : float, optional
    Proportion of data believed to be outliers. Used in conjunction with
    k_depth to determine the number of percentiles to draw. Defaults to
    0.007 as a proportion of outliers. Should be in range [0, 1].
showfliers : bool, optional
    If False, suppress the plotting of outliers.
{ax_in}
kwargs : key, value mappings
    Other keyword arguments are passed through to
    :meth:`matplotlib.axes.Axes.plot` and
    :meth:`matplotlib.axes.Axes.scatter`.

Returns
-----
{ax_out}

See Also
-----
{violinplot}
{boxplot}
{catplot}

Examples
-----
Draw a single horizontal boxen plot:
```

```
.. plot::
   :context: close-figs

>>> import seaborn as sns
>>> sns.set(style="whitegrid")
>>> tips = sns.load_dataset("tips")
>>> ax = sns.boxenplot(x=tips["total_bill"])

Draw a vertical boxen plot grouped by a categorical variable:

.. plot::
   :context: close-figs

>>> ax = sns.boxenplot(x="day", y="total_bill", data=tips)

Draw a letter value plot with nested grouping by two categorical variables:

.. plot::
   :context: close-figs

>>> ax = sns.boxenplot(x="day", y="total_bill", hue="smoker",
...                      data=tips, palette="Set3")

Draw a boxen plot with nested grouping when some bins are empty:

.. plot::
   :context: close-figs

>>> ax = sns.boxenplot(x="day", y="total_bill", hue="time",
...                      data=tips, linewidth=2.5)

Control box order by passing an explicit order:

.. plot::
   :context: close-figs

>>> ax = sns.boxenplot(x="time", y="tip", data=tips,
...                      order=["Dinner", "Lunch"])

Draw a boxen plot for each numeric variable in a DataFrame:

.. plot::
   :context: close-figs

>>> iris = sns.load_dataset("iris")
>>> ax = sns.boxenplot(data=iris, orient="h", palette="Set2")

Use :func:`stripplot` to show the datapoints on top of the boxes:

.. plot::
   :context: close-figs

>>> ax = sns.boxenplot(x="day", y="total_bill", data=tips)
>>> ax = sns.stripplot(x="day", y="total_bill", data=tips,
...                      size=4, color="gray")

Use :func:`catplot` to combine :func:`boxenplot` and a :class:`FacetGrid`. This allows grouping within additional categorical variables. Using :func:`catplot` is safer than using :class:`FacetGrid` directly, as it ensures synchronization of variable order across facets:
```

```

>>> g = sns.catplot(x="sex", y="total_bill",
...                   hue="smoker", col="time",
...                   data=tips, kind="boxen",
...                   height=4, aspect=.7);

""").format(**_categorical_docs)

def stripplot(x=None, y=None, hue=None, data=None, order=None, hue_order=None,
              jitter=True, dodge=False, orient=None, color=None, palette=None,
              size=5, edgecolor="gray", linewidth=0, ax=None, **kwargs):
    if "split" in kwargs:
        dodge = kwargs.pop("split")
        msg = "The `split` parameter has been renamed to `dodge`."
        warnings.warn(msg, UserWarning)

    plotter = _StripPlotter(x, y, hue, data, order, hue_order,
                           jitter, dodge, orient, color, palette)
    if ax is None:
        ax = plt.gca()

    kwargs.setdefault("zorder", 3)
    size = kwargs.get("s", size)
    if linewidth is None:
        linewidth = size / 10
    if edgecolor == "gray":
        edgecolor = plotter.gray
    kwargs.update(dict(s=size ** 2,
                       edgecolor=edgecolor,
                       linewidth=linewidth))

    plotter.plot(ax, kwargs)
    return ax

```

`stripplot.__doc__` = dedent("""\n Draw a scatterplot where one variable is categorical.

A strip plot can be drawn on its own, but it is also a good complement to a box or violin plot in cases where you want to show all observations along with some representation of the underlying distribution.

{main_api_narrative}

{categorical_narrative}

Parameters

{input_params}

{categorical_data}

{order_vars}

jitter : float, ``True``/``1`` is special-cased, optional

Amount of jitter (only along the categorical axis) to apply. This can be useful when you have many points and they overlap, so that it is easier to see the distribution. You can specify the amount of jitter (half the width of the uniform random variable support), or just use ``True`` for a good default.

dodge : bool, optional

When using ``hue`` nesting, setting this to ``True`` will separate the strips for different hue levels along the categorical axis.

Otherwise, the points for each level will be plotted on top of

```
    each other.
{orient}
{color}
{palette}
size : float, optional
    Radius of the markers, in points.
edgecolor : matplotlib color, "gray" is special-cased, optional
    Color of the lines around each point. If you pass ``"gray"`` , the
    brightness is determined by the color palette used for the body
    of the points.
{linewidth}
{ax_in}
kwargs : key, value mappings
    Other keyword arguments are passed through to
    :meth:`matplotlib.axes.Axes.scatter`.
```

Returns

```
{ax_out}
```

See Also

```
{swarmplot}
{boxplot}
{violinplot}
{catplot}
```

Examples

Draw a single horizontal strip plot:

```
.. plot::
   :context: close-figs

>>> import seaborn as sns
>>> sns.set(style="whitegrid")
>>> tips = sns.load_dataset("tips")
>>> ax = sns.stripplot(x=tips["total_bill"])
```

Group the strips by a categorical variable:

```
.. plot::
   :context: close-figs

>>> ax = sns.stripplot(x="day", y="total_bill", data=tips)
```

Use a smaller amount of jitter:

```
.. plot::
   :context: close-figs

>>> ax = sns.stripplot(x="day", y="total_bill", data=tips, jitter=0.05)
```

Draw horizontal strips:

```
.. plot::
   :context: close-figs

>>> ax = sns.stripplot(x="total_bill", y="day", data=tips)
```

Draw outlines around the points:

```

.. plot::
   :context: close-figs

>>> ax = sns.stripplot(x="total_bill", y="day", data=tips,
...                      linewidth=1)

Nest the strips within a second categorical variable:

.. plot::
   :context: close-figs

>>> ax = sns.stripplot(x="sex", y="total_bill", hue="day", data=tips)

Draw each level of the ``hue`` variable at different locations on the
major categorical axis:

.. plot::
   :context: close-figs

>>> ax = sns.stripplot(x="day", y="total_bill", hue="smoker",
...                      data=tips, palette="Set2", dodge=True)

Control strip order by passing an explicit order:

.. plot::
   :context: close-figs

>>> ax = sns.stripplot(x="time", y="tip", data=tips,
...                      order=["Dinner", "Lunch"])

Draw strips with large points and different aesthetics:

.. plot::
   :context: close-figs

>>> ax = sns.stripplot("day", "total_bill", "smoker", data=tips,
...                      palette="Set2", size=20, marker="D",
...                      edgecolor="gray", alpha=.25)

Draw strips of observations on top of a box plot:

.. plot::
   :context: close-figs

>>> import numpy as np
>>> ax = sns.boxplot(x="tip", y="day", data=tips, whis=np.inf)
>>> ax = sns.stripplot(x="tip", y="day", data=tips, color=".3")

Draw strips of observations on top of a violin plot:

.. plot::
   :context: close-figs

>>> ax = sns.violinplot(x="day", y="total_bill", data=tips,
...                        inner=None, color=".8")
>>> ax = sns.stripplot(x="day", y="total_bill", data=tips)

Use :func:`catplot` to combine a :func:`stripplot` and a
:class:`FacetGrid`. This allows grouping within additional categorical
variables. Using :func:`catplot` is safer than using :class:`FacetGrid`
directly, as it ensures synchronization of variable order across facets:

.. plot::

```

```

:context: close-figs

>>> g = sns.catplot(x="sex", y="total_bill",
...                     hue="smoker", col="time",
...                     data=tips, kind="strip",
...                     height=4, aspect=.7);

""").format(**_categorical_docs)

def swarmplot(x=None, y=None, hue=None, data=None, order=None, hue_order=None,
              dodge=False, orient=None, color=None, palette=None,
              size=5, edgecolor="gray", linewidth=0, ax=None, **kwargs):

    if "split" in kwargs:
        dodge = kwargs.pop("split")
        msg = "The `split` parameter has been renamed to `dodge`."
        warnings.warn(msg, UserWarning)

    plotter = _SwarmPlotter(x, y, hue, data, order, hue_order,
                           dodge, orient, color, palette)
    if ax is None:
        ax = plt.gca()

    kwargs.setdefault("zorder", 3)
    size = kwargs.get("s", size)
    if linewidth is None:
        linewidth = size / 10
    if edgecolor == "gray":
        edgecolor = plotter.gray
    kwargs.update(dict(s=size ** 2,
                       edgecolor=edgecolor,
                       linewidth=linewidth))

    plotter.plot(ax, kwargs)
    return ax

```

swarmplot.__doc__ = dedent("""\n Draw a categorical scatterplot with non-overlapping points.\n\n This function is similar to :func:`stripplot`, but the points are adjusted\n (only along the categorical axis) so that they don't overlap. This gives a\n better representation of the distribution of values, but it does not scale\n well to large numbers of observations. This style of plot is sometimes\n called a "beeswarm".\n\n A swarm plot can be drawn on its own, but it is also a good complement\n to a box or violin plot in cases where you want to show all observations\n along with some representation of the underlying distribution.\n\n Arranging the points properly requires an accurate transformation between\n data and point coordinates. This means that non-default axis limits must\n be set *before* drawing the plot.\n\n {main_api_narrative}\n\n {categorical_narrative}\n\n Parameters\n -----"\n {input_params}\n {categorical_data}\n

```
{order_vars}
dodge : bool, optional
    When using ``hue`` nesting, setting this to ``True`` will separate
    the strips for different hue levels along the categorical axis.
    Otherwise, the points for each level will be plotted in one swarm.
{orient}
{color}
{palette}
size : float, optional
    Radius of the markers, in points.
edgecolor : matplotlib color, "gray" is special-cased, optional
    Color of the lines around each point. If you pass ``"gray"``, the
    brightness is determined by the color palette used for the body
    of the points.
{linewidth}
{ax_in}
kwargs : key, value mappings
    Other keyword arguments are passed through to
    :meth:`matplotlib.axes.Axes.scatter`.
```

Returns

{ax_out}

See Also

{boxplot}
{violinplot}
{stripplot}
{catplot}

Examples

Draw a single horizontal swarm plot:

```
.. plot::
   :context: close-figs

>>> import seaborn as sns
>>> sns.set(style="whitegrid")
>>> tips = sns.load_dataset("tips")
>>> ax = sns.swarmplot(x=tips["total_bill"])
```

Group the swarms by a categorical variable:

```
.. plot::
   :context: close-figs

>>> ax = sns.swarmplot(x="day", y="total_bill", data=tips)
```

Draw horizontal swarms:

```
.. plot::
   :context: close-figs

>>> ax = sns.swarmplot(x="total_bill", y="day", data=tips)
```

Color the points using a second categorical variable:

```
.. plot::
   :context: close-figs
```

```

>>> ax = sns.swarmplot(x="day", y="total_bill", hue="sex", data=tips)

Split each level of the ``hue`` variable along the categorical axis:

.. plot::
   :context: close-figs

>>> ax = sns.swarmplot(x="day", y="total_bill", hue="smoker",
...                      data=tips, palette="Set2", dodge=True)

Control swarm order by passing an explicit order:

.. plot::
   :context: close-figs

>>> ax = sns.swarmplot(x="time", y="tip", data=tips,
...                      order=["Dinner", "Lunch"])

Plot using larger points:

.. plot::
   :context: close-figs

>>> ax = sns.swarmplot(x="time", y="tip", data=tips, size=6)

Draw swarms of observations on top of a box plot:

.. plot::
   :context: close-figs

>>> ax = sns.boxplot(x="tip", y="day", data=tips, whis=np.inf)
>>> ax = sns.swarmplot(x="tip", y="day", data=tips, color=".2")

Draw swarms of observations on top of a violin plot:

.. plot::
   :context: close-figs

>>> ax = sns.violinplot(x="day", y="total_bill", data=tips, inner=None)
>>> ax = sns.swarmplot(x="day", y="total_bill", data=tips,
...                      color="white", edgecolor="gray")

Use :func:`catplot` to combine a :func:`swarmplot` and a
:class:`FacetGrid`. This allows grouping within additional categorical
variables. Using :func:`catplot` is safer than using :class:`FacetGrid`
directly, as it ensures synchronization of variable order across facets:

.. plot::
   :context: close-figs

>>> g = sns.catplot(x="sex", y="total_bill",
...                    hue="smoker", col="time",
...                    data=tips, kind="swarm",
...                    height=4, aspect=.7);

""").format(**_categorical_docs)

def barplot(x=None, y=None, hue=None, data=None, order=None, hue_order=None,
           estimator=np.mean, ci=95, n_boot=1000, units=None, seed=None,
           orient=None, color=None, palette=None, saturation=.75,
           errcolor=".26", errwidth=None, caps=None, dodge=True,
           ax=None, **kwargs):

```

```
plotter = _BarPlotter(x, y, hue, data, order, hue_order,
                      estimator, ci, n_boot, units, seed,
                      orient, color, palette, saturation,
                      errcolor, erewidth, capsizes, dodge)

if ax is None:
    ax = plt.gca()

plotter.plot(ax, kwargs)
return ax

barplot.__doc__ = dedent("""\
Show point estimates and confidence intervals as rectangular bars.

A bar plot represents an estimate of central tendency for a numeric
variable with the height of each rectangle and provides some indication of
the uncertainty around that estimate using error bars. Bar plots include 0
in the quantitative axis range, and they are a good choice when 0 is a
meaningful value for the quantitative variable, and you want to make
comparisons against it.

For datasets where 0 is not a meaningful value, a point plot will allow you
to focus on differences between levels of one or more categorical
variables.

It is also important to keep in mind that a bar plot shows only the mean
(or other estimator) value, but in many cases it may be more informative to
show the distribution of values at each level of the categorical variables.
In that case, other approaches such as a box or violin plot may be more
appropriate.

{main_api_narrative}

{categorical_narrative}

Parameters
-----
{input_params}
{categorical_data}
{order_vars}
{stat_api_params}
{orient}
{color}
{palette}
{saturation}
errcolor : matplotlib color
    Color for the lines that represent the confidence interval.
{erewidth}
{capsizes}
{dodge}
{ax_in}
kwargs : key, value mappings
    Other keyword arguments are passed through to
    :meth:`matplotlib.axes.Axes.bar`.

Returns
-----
{ax_out}

See Also
-----
```

```
{countplot}  
{pointplot}  
{catplot}
```

Examples

Draw a set of vertical bar plots grouped by a categorical variable:

```
.. plot::  
    :context: close-figs  
  
>>> import seaborn as sns  
>>> sns.set(style="whitegrid")  
>>> tips = sns.load_dataset("tips")  
>>> ax = sns.barplot(x="day", y="total_bill", data=tips)
```

Draw a set of vertical bars with nested grouping by a two variables:

```
.. plot::  
    :context: close-figs  
  
>>> ax = sns.barplot(x="day", y="total_bill", hue="sex", data=tips)
```

Draw a set of horizontal bars:

```
.. plot::  
    :context: close-figs  
  
>>> ax = sns.barplot(x="tip", y="day", data=tips)
```

Control bar order by passing an explicit order:

```
.. plot::  
    :context: close-figs  
  
>>> ax = sns.barplot(x="time", y="tip", data=tips,  
...                     order=["Dinner", "Lunch"])
```

Use median as the estimate of central tendency:

```
.. plot::  
    :context: close-figs  
  
>>> from numpy import median  
>>> ax = sns.barplot(x="day", y="tip", data=tips, estimator=median)
```

Show the standard error of the mean with the error bars:

```
.. plot::  
    :context: close-figs  
  
>>> ax = sns.barplot(x="day", y="tip", data=tips, ci=68)
```

Show standard deviation of observations instead of a confidence interval:

```
.. plot::  
    :context: close-figs  
  
>>> ax = sns.barplot(x="day", y="tip", data=tips, ci="sd")
```

Add "caps" to the error bars:


```
        markers, linestyles, dodge, join, scale,
        orient, color, palette, errwidth, capsize)

if ax is None:
    ax = plt.gca()

plotter.plot(ax)
return ax

pointplot.__doc__ = dedent("""\
Show point estimates and confidence intervals using scatter plot glyphs.

A point plot represents an estimate of central tendency for a numeric
variable by the position of scatter plot points and provides some
indication of the uncertainty around that estimate using error bars.

Point plots can be more useful than bar plots for focusing comparisons
between different levels of one or more categorical variables. They are
particularly adept at showing interactions: how the relationship between
levels of one categorical variable changes across levels of a second
categorical variable. The lines that join each point from the same ``hue``
level allow interactions to be judged by differences in slope, which is
easier for the eyes than comparing the heights of several groups of points
or bars.

It is important to keep in mind that a point plot shows only the mean (or
other estimator) value, but in many cases it may be more informative to
show the distribution of values at each level of the categorical variables.
In that case, other approaches such as a box or violin plot may be more
appropriate.

{main_api_narrative}

{categorical_narrative}

Parameters
-----
{input_params}
{categorical_data}
{order_vars}
{stat_api_params}
markers : string or list of strings, optional
    Markers to use for each of the ``hue`` levels.
linestyles : string or list of strings, optional
    Line styles to use for each of the ``hue`` levels.
dodge : bool or float, optional
    Amount to separate the points for each level of the ``hue`` variable
    along the categorical axis.
join : bool, optional
    If ``True``, lines will be drawn between point estimates at the same
    ``hue`` level.
scale : float, optional
    Scale factor for the plot elements.
{orient}
{color}
{palette}
{errwidth}
{capsize}
{ax_in}

Returns
-----
```

A point plot represents an estimate of central tendency for a numeric variable by the position of scatter plot points and provides some indication of the uncertainty around that estimate using error bars.

Point plots can be more useful than bar plots for focusing comparisons between different levels of one or more categorical variables. They are particularly adept at showing interactions: how the relationship between levels of one categorical variable changes across levels of a second categorical variable. The lines that join each point from the same ``hue`` level allow interactions to be judged by differences in slope, which is easier for the eyes than comparing the heights of several groups of points or bars.

It is important to keep in mind that a point plot shows only the mean (or other estimator) value, but in many cases it may be more informative to show the distribution of values at each level of the categorical variables. In that case, other approaches such as a box or violin plot may be more appropriate.

```
{main_api_narrative}

{categorical_narrative}

Parameters
-----
{input_params}
{categorical_data}
{order_vars}
{stat_api_params}
markers : string or list of strings, optional
    Markers to use for each of the ``hue`` levels.
linestyles : string or list of strings, optional
    Line styles to use for each of the ``hue`` levels.
dodge : bool or float, optional
    Amount to separate the points for each level of the ``hue`` variable
    along the categorical axis.
join : bool, optional
    If ``True``, lines will be drawn between point estimates at the same
    ``hue`` level.
scale : float, optional
    Scale factor for the plot elements.
{orient}
{color}
{palette}
{errwidth}
{capsize}
{ax_in}

Returns
-----
```

```
{ax_out}
```

See Also

```
{barplot}
{catplot}
```

Examples

Draw a set of vertical point plots grouped by a categorical variable:

```
.. plot::  
    :context: close-figs  
  
    >>> import seaborn as sns  
    >>> sns.set(style="darkgrid")  
    >>> tips = sns.load_dataset("tips")  
    >>> ax = sns.pointplot(x="time", y="total_bill", data=tips)
```

Draw a set of vertical points with nested grouping by a two variables:

```
.. plot::  
    :context: close-figs  
  
    >>> ax = sns.pointplot(x="time", y="total_bill", hue="smoker",  
    ...                         data=tips)
```

Separate the points for different hue levels along the categorical axis:

```
.. plot::  
    :context: close-figs  
  
    >>> ax = sns.pointplot(x="time", y="total_bill", hue="smoker",  
    ...                         data=tips, dodge=True)
```

Use a different marker and line style for the hue levels:

```
.. plot::  
    :context: close-figs  
  
    >>> ax = sns.pointplot(x="time", y="total_bill", hue="smoker",  
    ...                         data=tips,  
    ...                         markers=["o", "x"],  
    ...                         linestyles=[ "-", "--"])
```

Draw a set of horizontal points:

```
.. plot::  
    :context: close-figs  
  
    >>> ax = sns.pointplot(x="tip", y="day", data=tips)
```

Don't draw a line connecting each point:

```
.. plot::  
    :context: close-figs  
  
    >>> ax = sns.pointplot(x="tip", y="day", data=tips, join=False)
```

Use a different color for a single-layer plot:

```
.. plot::
```

```
:context: close-figs

>>> ax = sns.pointplot("time", y="total_bill", data=tips,
...                      color="#bb3f3f")
```

Use a different color palette for the points:

```
.. plot::
   :context: close-figs

>>> ax = sns.pointplot(x="time", y="total_bill", hue="smoker",
...                      data=tips, palette="Set2")
```

Control point order by passing an explicit order:

```
.. plot::
   :context: close-figs

>>> ax = sns.pointplot(x="time", y="tip", data=tips,
...                      order=["Dinner", "Lunch"])
```

Use median as the estimate of central tendency:

```
.. plot::
   :context: close-figs

>>> from numpy import median
>>> ax = sns.pointplot(x="day", y="tip", data=tips, estimator=median)
```

Show the standard error of the mean with the error bars:

```
.. plot::
   :context: close-figs

>>> ax = sns.pointplot(x="day", y="tip", data=tips, ci=68)
```

Show standard deviation of observations instead of a confidence interval:

```
.. plot::
   :context: close-figs

>>> ax = sns.pointplot(x="day", y="tip", data=tips, ci="sd")
```

Add "caps" to the error bars:

```
.. plot::
   :context: close-figs

>>> ax = sns.pointplot(x="day", y="tip", data=tips, caps=.2)
```

Use `:func:`catplot`` to combine a `:func:`pointplot`` and a `:class:`FacetGrid``. This allows grouping within additional categorical variables. Using `:func:`catplot`` is safer than using `:class:`FacetGrid`` directly, as it ensures synchronization of variable order across facets:

```
.. plot::
   :context: close-figs

>>> g = sns.catplot(x="sex", y="total_bill",
...                   hue="smoker", col="time",
...                   data=tips, kind="point",
...                   dodge=True,
...                   height=4, aspect=.7);
```

```
""").format(**_categorical_docs)

def countplot(x=None, y=None, hue=None, data=None, order=None, hue_order=None,
              orient=None, color=None, palette=None, saturation=.75,
              dodge=True, ax=None, **kwargs):

    estimator = len
    ci = None
    n_boot = 0
    units = None
    seed = None
    errcolor = None
    errwidth = None
    capsiz = None

    if x is None and y is not None:
        orient = "h"
        x = y
    elif y is None and x is not None:
        orient = "v"
        y = x
    elif x is not None and y is not None:
        raise TypeError("Cannot pass values for both `x` and `y`")
    else:
        raise TypeError("Must pass values for either `x` or `y`")

    plotter = _BarPlotter(x, y, hue, data, order, hue_order,
                          estimator, ci, n_boot, units, seed,
                          orient, color, palette, saturation,
                          errcolor, errwidth, capsiz, dodge)

    plotter.value_label = "count"

    if ax is None:
        ax = plt.gca()

    plotter.plot(ax, kwargs)
    return ax
```

countplot.__doc__ = dedent("""\n Show the counts of observations in each categorical bin using bars.

A count plot can be thought of as a histogram across a categorical, instead of quantitative, variable. The basic API and options are identical to those for :func:`barplot`, so you can compare counts across nested variables.

{main_api_narrative}

{categorical_narrative}

Parameters

{input_params}
{categorical_data}
{order_vars}
{orient}
{color}
{palette}
{saturation}
{dodge}

```
{ax_in}
    kwards : key, value mappings
        Other keyword arguments are passed through to
        :meth:`matplotlib.axes.Axes.bar`.

Returns
-----
{ax_out}

See Also
-----
{barplot}
{catplot}

Examples
-----
Show value counts for a single categorical variable:

.. plot::
   :context: close-figs

   >>> import seaborn as sns
   >>> sns.set(style="darkgrid")
   >>> titanic = sns.load_dataset("titanic")
   >>> ax = sns.countplot(x="class", data=titanic)

Show value counts for two categorical variables:

.. plot::
   :context: close-figs

   >>> ax = sns.countplot(x="class", hue="who", data=titanic)

Plot the bars horizontally:

.. plot::
   :context: close-figs

   >>> ax = sns.countplot(y="class", hue="who", data=titanic)

Use a different color palette:

.. plot::
   :context: close-figs

   >>> ax = sns.countplot(x="who", data=titanic, palette="Set3")

Use :meth:`matplotlib.axes.Axes.bar` parameters to control the style.

.. plot::
   :context: close-figs

   >>> ax = sns.countplot(x="who", data=titanic,
   ...                      facecolor=(0, 0, 0, 0),
   ...                      linewidth=5,
   ...                      edgecolor=sns.color_palette("dark", 3))

Use :func:`catplot` to combine a :func:`countplot` and a
:class:`FacetGrid`. This allows grouping within additional categorical
variables. Using :func:`catplot` is safer than using :class:`FacetGrid`
directly, as it ensures synchronization of variable order across facets:
```

```

.. plot::
   :context: close-figs

    >>> g = sns.catplot(x="class", hue="who", col="survived",
...                      data=titanic, kind="count",
...                      height=4, aspect=.7);

""").format(**_categorical_docs)

def factorplot(*args, **kwargs):
    """Deprecated; please use `catplot` instead."""
    msg = (
        "The `factorplot` function has been renamed to `catplot`. The "
        "original name will be removed in a future release. Please update "
        "your code. Note that the default `kind` in `factorplot` ('point') "
        "has changed `strip` in `catplot`."
    )
    warnings.warn(msg)

    if "size" in kwargs:
        kwargs["height"] = kwargs.pop("size")
        msg = ("The `size` parameter has been renamed to `height`; "
               "please update your code.")
        warnings.warn(msg, UserWarning)

    kwargs.setdefault("kind", "point")

    return catplot(*args, **kwargs)

def catplot(x=None, y=None, hue=None, data=None, col=None,
           col_wrap=None, estimator=np.mean, ci=95, n_boot=1000,
           units=None, seed=None, order=None, hue_order=None, row_order=None,
           col_order=None, kind="strip", height=5, aspect=1,
           orient=None, color=None, palette=None,
           legend=True, legend_out=True, sharex=True, sharey=True,
           margin_titles=False, facet_kws=None, **kwargs):

    # Handle deprecations
    if "size" in kwargs:
        height = kwargs.pop("size")
        msg = ("The `size` parameter has been renamed to `height`; "
               "please update your code.")
        warnings.warn(msg, UserWarning)

    # Determine the plotting function
    try:
        plot_func = globals()[kind + "plot"]
    except KeyError:
        err = "Plot kind '{}' is not recognized".format(kind)
        raise ValueError(err)

    # Alias the input variables to determine categorical order and palette
    # correctly in the case of a count plot
    if kind == "count":
        if x is None and y is not None:
            x_, y_, orient = y, y, "h"
        elif y is None and x is not None:
            x_, y_, orient = x, x, "v"
        else:
            raise ValueError("Either `x` or `y` must be None for count plots")

```

```

else:
    x_, y_ = x, y

# Check for attempt to plot onto specific axes and warn
if "ax" in kwargs:
    msg = ("catplot is a figure-level function and does not accept "
           "target axes. You may wish to try {}".format(kind + "plot"))
    warnings.warn(msg, UserWarning)
    kwargs.pop("ax")

# Determine the order for the whole dataset, which will be used in all
# facets to ensure representation of all data in the final plot
p = _CategoricalPlotter()
p.establish_variables(x_, y_, hue, data, orient, order, hue_order)
order = p.group_names
hue_order = p.hue_names

# Determine the palette to use
# (FacetGrid will pass a value for ``color`` to the plotting function
# so we need to define ``palette`` to get default behavior for the
# categorical functions
p.establish_colors(color, palette, 1)
if kind != "point" or hue is not None:
    palette = p.colors

# Determine keyword arguments for the facets
facet_kws = {} if facet_kws is None else facet_kws
facet_kws.update(
    data=data, row=row, col=col,
    row_order=row_order, col_order=col_order,
    col_wrap=col_wrap, height=height, aspect=aspect,
    sharex=sharex, sharey=sharey,
    legend_out=legend_out, margin_titles=margin_titles,
    dropna=False,
)

# Determine keyword arguments for the plotting function
plot_kws = dict(
    order=order, hue_order=hue_order,
    orient=orient, color=color, palette=palette,
)
plot_kws.update(kwargs)

if kind in ["bar", "point"]:
    plot_kws.update(
        estimator=estimator, ci=ci, n_boot=n_boot, units=units, seed=seed,
    )

# Initialize the facets
g = FacetGrid(**facet_kws)

# Draw the plot onto the facets
g.map_dataframe(plot_func, x, y, hue, **plot_kws)

# Special case axis labels for a count type plot
if kind == "count":
    if x is None:
        g.set_axis_labels(x_var="count")
    if y is None:
        g.set_axis_labels(y_var="count")

if legend and (hue is not None) and (hue not in [x, row, col]):
    hue_order = list(map(utils.to_utf8, hue_order))

```

```
    g.add_legend(title=hue, label_order=hue_order)

return g

catplot.__doc__ = dedent("""\
Figure-level interface for drawing categorical plots onto a
:class:`FacetGrid`.
```

This function provides access to several axes-level functions that show the relationship between a numerical and one or more categorical variables using one of several visual representations. The ``kind`` parameter selects the underlying axes-level function to use:

Categorical scatterplots:

- :func:`stripplot` (with ``kind="strip"``; the default)
- :func:`swarmplot` (with ``kind="swarm"``)

Categorical distribution plots:

- :func:`boxplot` (with ``kind="box"``)
- :func:`violinplot` (with ``kind="violin"``)
- :func:`boxenplot` (with ``kind="boxen"``)

Categorical estimate plots:

- :func:`pointplot` (with ``kind="point"``)
- :func:`barplot` (with ``kind="bar"``)
- :func:`countplot` (with ``kind="count"``)

Extra keyword arguments are passed to the underlying function, so you should refer to the documentation for each to see kind-specific options.

Note that unlike when using the axes-level functions directly, data must be passed in a long-form DataFrame with variables specified by passing strings to ``x``, ``y``, ``hue``, etc.

As in the case with the underlying plot functions, if variables have a ``categorical`` data type, the levels of the categorical variables, and their order will be inferred from the objects. Otherwise you may have to use alter the dataframe sorting or use the function parameters (``orient``, ``order``, ``hue_order``, etc.) to set up the plot correctly.

{categorical_narrative}

After plotting, the :class:`FacetGrid` with the plot is returned and can be used directly to tweak supporting plot details or add other layers.

Parameters

```
{string_input_params}
{long_form_data}
row, col : names of variables in ``data``, optional
    Categorical variables that will determine the faceting of the grid.
{col_wrap}
{stat_api_params}
{order_vars}
row_order, col_order : lists of strings, optional
    Order to organize the rows and/or columns of the grid in, otherwise the
    orders are inferred from the data objects.
kind : string, optional
    The kind of plot to draw (corresponds to the name of a categorical
```

```
plotting function. Options are: "point", "bar", "strip", "swarm",
"box", "violin", or "boxen".
{height}
{aspect}
{orient}
{color}
{palette}
legend : bool, optional
    If ``True`` and there is a ``hue`` variable, draw a legend on the plot.
{legend_out}
{share_xy}
{margin_titles}
facet_kws : dict, optional
    Dictionary of other keyword arguments to pass to :class:`FacetGrid`.
kwargs : key, value pairings
    Other keyword arguments are passed through to the underlying plotting
    function.
```

Returns

```
-----
g : :class:`FacetGrid`
    Returns the :class:`FacetGrid` object with the plot on it for further
    tweaking.
```

Examples

```
Draw a single facet to use the :class:`FacetGrid` legend placement:
```

```
.. plot::
   :context: close-figs

>>> import seaborn as sns
>>> sns.set(style="ticks")
>>> exercise = sns.load_dataset("exercise")
>>> g = sns.catplot(x="time", y="pulse", hue="kind", data=exercise)
```

```
Use a different plot kind to visualize the same data:
```

```
.. plot::
   :context: close-figs

>>> g = sns.catplot(x="time", y="pulse", hue="kind",
...                   data=exercise, kind="violin")
```

```
Facet along the columns to show a third categorical variable:
```

```
.. plot::
   :context: close-figs

>>> g = sns.catplot(x="time", y="pulse", hue="kind",
...                   col="diet", data=exercise)
```

```
Use a different height and aspect ratio for the facets:
```

```
.. plot::
   :context: close-figs

>>> g = sns.catplot(x="time", y="pulse", hue="kind",
...                   col="diet", data=exercise,
...                   height=5, aspect=.8)
```

```
Make many column facets and wrap them into the rows of the grid:
```

```
.. plot::
   :context: close-figs

>>> titanic = sns.load_dataset("titanic")
>>> g = sns.catplot("alive", col="deck", col_wrap=4,
...                   data=titanic[titanic.deck.notnull()],
...                   kind="count", height=2.5, aspect=.8)

Plot horizontally and pass other keyword arguments to the plot function:

.. plot::
   :context: close-figs

>>> g = sns.catplot(x="age", y="embark_town",
...                   hue="sex", row="class",
...                   data=titanic[titanic.embark_town.notnull()],
...                   orient="h", height=2, aspect=3, palette="Set3",
...                   kind="violin", dodge=True, cut=0, bw=.2)

Use methods on the returned :class:`FacetGrid` to tweak the presentation:

.. plot::
   :context: close-figs

>>> g = sns.catplot(x="who", y="survived", col="class",
...                   data=titanic, saturation=.5,
...                   kind="bar", ci=None, aspect=.6)
>>> (g.set_axis_labels("", "Survival Rate")
...     .set_xticklabels(["Men", "Women", "Children"])
...     .set_titles("{col_name} {col_var}")
...     .set(ylim=(0, 1))
...     .despine(left=True)) #doctest: +ELLIPSIS
<seaborn.axisgrid.FacetGrid object at 0x...>

""").format(**_categorical_docs)
```

<https://github.com/pandas-dev/pandas/blob/master/pandas/core/indexing.py>

```
from typing import TYPE_CHECKING, Hashable, List, Tuple, Union

import numpy as np

from pandas._libs.indexing import _NDFrameIndexerBase
from pandas._libs.lib import item_from_zerodim
from pandas.errors import AbstractMethodError
from pandas.util._decorators import doc

from pandas.core.dtypes.common import (
    is_hashable,
    is_integer,
    is_iterator,
    is_list_like,
    is_numeric_dtype,
    is_object_dtype,
    is_scalar,
    is_sequence,
)
from pandas.core.dtypes.concat import concat_compat
from pandas.core.dtypes.generic import ABCDataFrame, ABCMultiIndex, ABCSeries
from pandas.core.dtypes.missing import _infer_fill_value, isna

import pandas.core.common as com
from pandas.core.indexers import (
    check_array_indexer,
    is_list_like_indexer,
    length_of_indexer,
)
from pandas.core.indexes.api import Index, InvalidIndexError

if TYPE_CHECKING:
    from pandas import DataFrame  # noqa:F401

# "null slice"
_NS = slice(None, None)

# the public IndexSlicerMaker
class _IndexSlice:
    """
    Create an object to more easily perform multi-index slicing.

    See Also
    -----
    MultiIndex.remove_unused_levels : New MultiIndex with no unused levels.

    Notes
    -----
    See :ref:`Defined Levels <advanced.shown_levels>` for further info on slicing a MultiIndex.

    Examples
    -----
    >>> midx = pd.MultiIndex.from_product([['A0', 'A1'], ['B0', 'B1', 'B2', 'B3']])

```

```
>>> columns = ['foo', 'bar']
>>> dfmi = pd.DataFrame(np.arange(16).reshape((len(midx), len(columns))),
                           index=midx, columns=columns)
```

Using the default slice command:

```
>>> dfmi.loc[(slice(None), slice('B0', 'B1')), :]
      foo   bar
A0  B0    0    1
     B1    2    3
A1  B0    8    9
     B1   10   11
```

Using the IndexSlice class for a more intuitive command:

```
>>> idx = pd.IndexSlice
>>> dfmi.loc[idx[:, 'B0':'B1'], :]
      foo   bar
A0  B0    0    1
     B1    2    3
A1  B0    8    9
     B1   10   11
```

```
"""
def __getitem__(self, arg):
    return arg
```

```
IndexSlice = _IndexSlice()
```

```
class IndexingError(Exception):
    pass
```

```
class IndexingMixin:
    """
    Mixin for adding .loc/.iloc/.at/.iat to Dataframes and Series.
    """
```

```
@property
def iloc(self) -> "_iLocIndexer":
    """
    Purely integer-location based indexing for selection by position.
```

```
```.iloc[]`` is primarily integer position based (from ``0`` to
``length-1`` of the axis), but may also be used with a boolean
array.
```

Allowed inputs are:

- An integer, e.g. ``5``.
- A list or array of integers, e.g. ``[4, 3, 0]``.
- A slice object with ints, e.g. ``1:7``.
- A boolean array.
- A ``callable`` function with one argument (the calling Series or
DataFrame) and that returns valid output for indexing (one of the above).
This is useful in method chains, when you don't have a reference to the
calling object, but would like to base your selection on some value.

```.iloc`` will raise ``IndexError`` if a requested indexer is
out-of-bounds, except *slice* indexers which allow out-of-bounds
indexing (this conforms with python/numpy *slice* semantics).

See more at :ref:`Selection by Position <indexing.integer>`.

See Also

DataFrame.iat : Fast integer location scalar accessor.
DataFrame.loc : Purely label-location based indexer for selection by label.
Series.iloc : Purely integer-location based indexing for
selection by position.

Examples

>>> mydict = [{‘a’: 1, ‘b’: 2, ‘c’: 3, ‘d’: 4},
... {‘a’: 100, ‘b’: 200, ‘c’: 300, ‘d’: 400},
... {‘a’: 1000, ‘b’: 2000, ‘c’: 3000, ‘d’: 4000 }]
>>> df = pd.DataFrame(mydict)
>>> df
a b c d
0 1 2 3 4
1 100 200 300 400
2 1000 2000 3000 4000

Indexing just the rows

With a scalar integer.

```
>>> type(df.iloc[0])  
<class ‘pandas.core.series.Series’>  
>>> df.iloc[0]  
a 1  
b 2  
c 3  
d 4  
Name: 0, dtype: int64
```

With a list of integers.

```
>>> df.iloc[[0]]  
a b c d  
0 1 2 3 4  
>>> type(df.iloc[[0]])  
<class ‘pandas.core.frame.DataFrame’>  
  
>>> df.iloc[[0, 1]]  
a b c d  
0 1 2 3 4  
1 100 200 300 400
```

With a `slice` object.

```
>>> df.iloc[:3]  
a b c d  
0 1 2 3 4  
1 100 200 300 400  
2 1000 2000 3000 4000
```

With a boolean mask the same length as the index.

```
>>> df.iloc[[True, False, True]]  
a b c d  
0 1 2 3 4  
2 1000 2000 3000 4000
```

```
With a callable, useful in method chains. The `x` passed to the ``lambda`` is the DataFrame being sliced. This selects the rows whose index label even.
```

```
>>> df.iloc[lambda x: x.index % 2 == 0]
      a      b      c      d
0      1      2      3      4
2    1000    2000    3000    4000

**Indexing both axes**
```

```
You can mix the indexer types for the index and columns. Use ``:`` to select the entire axis.
```

```
With scalar integers.
```

```
>>> df.iloc[0, 1]
2
```

```
With lists of integers.
```

```
>>> df.iloc[[0, 2], [1, 3]]
      b      d
0      2      4
2    2000    4000
```

```
With `slice` objects.
```

```
>>> df.iloc[1:3, 0:3]
      a      b      c
1    100    200    300
2   1000   2000   3000
```

```
With a boolean array whose length matches the columns.
```

```
>>> df.iloc[:, [True, False, True, False]]
      a      c
0      1      3
1    100    300
2   1000   3000
```

```
With a callable function that expects the Series or DataFrame.
```

```
>>> df.iloc[:, lambda df: [0, 2]]
      a      c
0      1      3
1    100    300
2   1000   3000
"""
return _iLocIndexer("iloc", self)
```

```
@property
def loc(self) -> "_LocIndexer":
    """
Access a group of rows and columns by label(s) or a boolean array.
```

```
``.loc[]`` is primarily label based, but may also be used with a boolean array.
```

```
Allowed inputs are:
```

- A single label, e.g. ``5`` or ``'a'``, (note that ``5`` is interpreted as a *label* of the index, and **never** as an

```

        integer position along the index).
- A list or array of labels, e.g. ``['a', 'b', 'c']``.
- A slice object with labels, e.g. ``'a':'f'``.

.. warning:: Note that contrary to usual python slices, **both** the
start and the stop are included

- A boolean array of the same length as the axis being sliced,
e.g. ``[True, False, True]``.
- A ``callable`` function with one argument (the calling Series or
DataFrame) and that returns valid output for indexing (one of the above)

See more at :ref:`Selection by Label <indexing.label>`
```

Raises

KeyError

If any items are not found.

See Also

DataFrame.at : Access a single value for a row/column label pair.
DataFrame.iloc : Access group of rows and columns by integer position(s).
DataFrame.xs : Returns a cross-section (row(s) or column(s)) from the
Series/DataFrame.
Series.loc : Access group of values using labels.

Examples

Getting values

```

>>> df = pd.DataFrame([[1, 2], [4, 5], [7, 8]],
...                     index=['cobra', 'viper', 'sidewinder'],
...                     columns=['max_speed', 'shield'])
>>> df
      max_speed  shield
cobra          1        2
viper          4        5
sidewinder     7        8
```

Single label. Note this returns the row as a Series.

```

>>> df.loc['viper']
max_speed    4
shield       5
Name: viper, dtype: int64
```

List of labels. Note using ``[]`` returns a DataFrame.

```

>>> df.loc[['viper', 'sidewinder']]
      max_speed  shield
viper          4        5
sidewinder     7        8
```

Single label for row and column

```

>>> df.loc['cobra', 'shield']
2
```

Slice with labels for row and single label for column. As mentioned
above, note that both the start and stop of the slice are included.

```

>>> df.loc['cobra':'viper', 'max_speed']
```

```

cobra    1
viper    4
Name: max_speed, dtype: int64

Boolean list with the same length as the row axis

>>> df.loc[[False, False, True]]
      max_speed  shield
sidewinder          7       8

Conditional that returns a boolean Series

>>> df.loc[df['shield'] > 6]
      max_speed  shield
sidewinder          7       8

Conditional that returns a boolean Series with column labels specified

>>> df.loc[df['shield'] > 6, ['max_speed']]
      max_speed
sidewinder          7

Callable that returns a boolean Series

>>> df.loc[lambda df: df['shield'] == 8]
      max_speed  shield
sidewinder          7       8

**Setting values**

Set value for all items matching the list of labels

>>> df.loc[['viper', 'sidewinder'], ['shield']] = 50
>>> df
      max_speed  shield
cobra          1       2
viper          4      50
sidewinder      7      50

Set value for an entire row

>>> df.loc['cobra'] = 10
>>> df
      max_speed  shield
cobra          10      10
viper          4       50
sidewinder      7       50

Set value for an entire column

>>> df.loc[:, 'max_speed'] = 30
>>> df
      max_speed  shield
cobra          30      10
viper          30      50
sidewinder      30      50

Set value for rows matching callable condition

>>> df.loc[df['shield'] > 35] = 0
>>> df
      max_speed  shield
cobra          30      10

```

```
viper          0      0
sidewinder     0      0
```

Getting values on a DataFrame with an index that has integer labels

Another example using integers for the index

```
>>> df = pd.DataFrame([[1, 2], [4, 5], [7, 8]],
...                   index=[7, 8, 9], columns=['max_speed', 'shield'])
>>> df
   max_speed  shield
7           1       2
8           4       5
9           7       8
```

Slice with integer labels for rows. As mentioned above, note that both the start and stop of the slice are included.

```
>>> df.loc[7:9]
   max_speed  shield
7           1       2
8           4       5
9           7       8
```

Getting values with a MultiIndex

A number of examples using a DataFrame with a MultiIndex

```
>>> tuples = [
...     ('cobra', 'mark i'), ('cobra', 'mark ii'),
...     ('sidewinder', 'mark i'), ('sidewinder', 'mark ii'),
...     ('viper', 'mark ii'), ('viper', 'mark iii')
... ]
>>> index = pd.MultiIndex.from_tuples(tuples)
>>> values = [[12, 2], [0, 4], [10, 20],
...             [1, 4], [7, 1], [16, 36]]
>>> df = pd.DataFrame(values, columns=['max_speed', 'shield'], index=index)
>>> df
                           max_speed  shield
cobra    mark i            12       2
              mark ii          0       4
sidewinder mark i            10      20
              mark ii           1       4
viper     mark ii            7       1
              mark iii          16      36
```

Single label. Note this returns a DataFrame with a single index.

```
>>> df.loc['cobra']
                           max_speed  shield
mark i            12       2
mark ii           0       4
```

Single index tuple. Note this returns a Series.

```
>>> df.loc[('cobra', 'mark ii')]
max_speed    0
shield      4
Name: (cobra, mark ii), dtype: int64
```

Single label for row and column. Similar to passing in a tuple, this returns a Series.

```
>>> df.loc['cobra', 'mark i']
max_speed    12
shield       2
Name: (cobra, mark i), dtype: int64

Single tuple. Note using ``[]`` returns a DataFrame.

>>> df.loc[['cobra', 'mark ii']]]
          max_speed  shield
cobra  mark ii      0       4

Single tuple for the index with a single label for the column

>>> df.loc[('cobra', 'mark i'), 'shield']
2

Slice from index tuple to single label

>>> df.loc[('cobra', 'mark i'):'viper']
          max_speed  shield
cobra  mark i      12       2
        mark ii      0       4
sidewinder  mark i      10      20
        mark ii      1       4
viper    mark ii      7       1
        mark iii     16      36

Slice from index tuple to index tuple

>>> df.loc[('cobra', 'mark i'):('viper', 'mark ii')]
          max_speed  shield
cobra  mark i      12       2
        mark ii      0       4
sidewinder  mark i      10      20
        mark ii      1       4
viper    mark ii      7       1
"""
return _LocIndexer("loc", self)

@property
def at(self) -> "_AtIndexer":
"""
Access a single value for a row/column label pair.

Similar to ``loc``, in that both provide label-based lookups. Use
``at`` if you only need to get or set a single value in a DataFrame
or Series.

Raises
-----
KeyError
    If 'label' does not exist in DataFrame.

See Also
-----
DataFrame.iat : Access a single value for a row/column pair by integer
    position.
DataFrame.loc : Access a group of rows and columns by label(s).
Series.at : Access a single value using a label.

Examples
-----
>>> df = pd.DataFrame([[0, 2, 3], [0, 4, 1], [10, 20, 30]],
```

```
...                                         index=[4, 5, 6], columns=['A', 'B', 'C'])
>>> df
   A    B    C
4  0    2    3
5  0    4    1
6 10   20   30

Get value at specified row/column pair
>>> df.at[4, 'B']
2

Set value at specified row/column pair
>>> df.at[4, 'B'] = 10
>>> df.at[4, 'B']
10

Get value within a Series
>>> df.loc[5].at['B']
4
"""
return _AtIndexer("at", self)

@property
def iat(self) -> "_iAtIndexer":
"""
Access a single value for a row/column pair by integer position.

Similar to ``iloc``, in that both provide integer-based lookups. Use
``iat`` if you only need to get or set a single value in a DataFrame
or Series.

Raises
-----
IndexError
    When integer position is out of bounds.

See Also
-----
DataFrame.at : Access a single value for a row/column label pair.
DataFrame.loc : Access a group of rows and columns by label(s).
DataFrame.iloc : Access a group of rows and columns by integer position(s).

Examples
-----
>>> df = pd.DataFrame([[0, 2, 3], [0, 4, 1], [10, 20, 30]],
...                      columns=['A', 'B', 'C'])
>>> df
   A    B    C
0  0    2    3
1  0    4    1
2 10   20   30

Get value at specified row/column pair
>>> df.iat[1, 2]
1

Set value at specified row/column pair
>>> df.iat[1, 2] = 10
```

```

>>> df.iat[1, 2]
10

Get value within a series

>>> df.loc[0].iat[1]
2
"""
    return _iAtIndexer("iat", self)

class _LocationIndexer(_NDFrameIndexerBase):
    _valid_types: str
    axis = None

    def __call__(self, axis=None):
        # we need to return a copy of ourselves
        new_self = type(self)(self.name, self.obj)

        if axis is not None:
            axis = self.obj._get_axis_number(axis)
        new_self.axis = axis
        return new_self

    def __getitem__(self, key):
        """
        Convert a potentially-label-based key into a positional indexer.
        """
        if self.name == "loc":
            self._ensure_listlike_indexer(key)

        if self.axis is not None:
            return self._convert_tuple(key, is_setter=True)

        ax = self.obj._get_axis(0)

        if isinstance(ax, ABCMultiIndex) and self.name != "iloc":
            try:
                return ax.get_loc(key)
            except (TypeError, KeyError, InvalidIndexError):
                # TypeError e.g. passed a bool
                pass

        if isinstance(key, tuple):
            try:
                return self._convert_tuple(key, is_setter=True)
            except IndexingError:
                pass

        if isinstance(key, range):
            return list(key)

        try:
            return self._convert_to_indexer(key, axis=0, is_setter=True)
        except TypeError as e:

            # invalid indexer type vs 'other' indexing errors
            if "cannot do" in str(e):
                raise
            elif "unhashable type" in str(e):
                raise
            raise IndexingError(key) from e

```

```

def _ensure_listlike_indexer(self, key, axis=None):
    """
    Ensure that a list-like of column labels are all present by adding them if
    they do not already exist.

    Parameters
    -----
    key : list-like of column labels
        Target labels.
    axis : key axis if known
    """
    column_axis = 1

    # column only exists in 2-dimensional DataFrame
    if self.ndim != 2:
        return

    if isinstance(key, tuple):
        # key may be a tuple if we are .loc
        # in that case, set key to the column part of key
        key = key[column_axis]
        axis = column_axis

    if (
        axis == column_axis
        and not isinstance(self.obj.columns, ABCMultiIndex)
        and is_list_like_indexer(key)
        and not com.is_bool_indexer(key)
        and all(is_hashable(k) for k in key)
    ):
        for k in key:
            if k not in self.obj:
                self.obj[k] = np.nan

def __setitem__(self, key, value):
    if isinstance(key, tuple):
        key = tuple(com.apply_if_callable(x, self.obj) for x in key)
    else:
        key = com.apply_if_callable(key, self.obj)
    indexer = self._get_setitem_indexer(key)
    self._has_valid_setitem_indexer(key)

    iloc = self if self.name == "iloc" else self.obj.iloc
    iloc._setitem_with_indexer(indexer, value)

def _validate_key(self, key, axis: int):
    """
    Ensure that key is valid for current indexer.

    Parameters
    -----
    key : scalar, slice or list-like
        Key requested.
    axis : int
        Dimension on which the indexing is being made.

    Raises
    -----
    TypeError
        If the key (or some element of it) has wrong type.
    IndexError
        If the key (or some element of it) is out of bounds.
    KeyError
    """

```

```

        If the key was not found.
    """
    raise AbstractMethodError(self)

def _has_valid_tuple(self, key: Tuple):
    """
    Check the key for valid keys across my indexer.
    """
    for i, k in enumerate(key):
        if i >= self.ndim:
            raise IndexingError("Too many indexers")
        try:
            self._validate_key(k, i)
        except ValueError as err:
            raise ValueError(
                "Location based indexing can only have "
                f"[{self._valid_types}] types"
            ) from err

def _is_nested_tuple_indexer(self, tup: Tuple) -> bool:
    """
    Returns
    -----
    bool
    """
    if any(isinstance(ax, ABCMultiIndex) for ax in self.obj.axes):
        return any(_nested_tuple(tup, ax) for ax in self.obj.axes)
    return False

def _convert_tuple(self, key, is_setter: bool = False):
    keyidx = []
    if self.axis is not None:
        axis = self.obj._get_axis_number(self.axis)
        for i in range(self.ndim):
            if i == axis:
                keyidx.append(
                    self._convert_to_indexer(key, axis=axis, is_setter=is_setter)
                )
            else:
                keyidx.append(slice(None))
    else:
        for i, k in enumerate(key):
            if i >= self.ndim:
                raise IndexingError("Too many indexers")
            idx = self._convert_to_indexer(k, axis=i, is_setter=is_setter)
            keyidx.append(idx)
    return tuple(keyidx)

def _getitem_tuple_same_dim(self, tup: Tuple):
    """
    Index with indexers that should return an object of the same dimension
    as self.obj.

    This is only called after a failed call to _getitem_lowerdim.
    """
    retval = self.obj
    for i, key in enumerate(tup):
        if com.is_null_slice(key):
            continue

            retval = getattr(retval, self.name)._getitem_axis(key, axis=i)
            # We should never have retval.ndim < self.ndim, as that should
            # be handled by the _getitem_lowerdim call above.

```

```

        assert retval.ndim == self.ndim

    return retval

def _getitem_lowerdim(self, tup: Tuple):

    # we can directly get the axis result since the axis is specified
    if self.axis is not None:
        axis = self.obj._get_axis_number(self.axis)
        return self._getitem_axis(tup, axis=axis)

    # we may have a nested tuples indexer here
    if self._is_nested_tuple_indexer(tup):
        return self._getitem_nested_tuple(tup)

    # we maybe be using a tuple to represent multiple dimensions here
    ax0 = self.obj._get_axis(0)
    # ...but iloc should handle the tuple as simple integer-location
    # instead of checking it as multiindex representation (GH 13797)
    if isinstance(ax0, ABCMultiIndex) and self.name != "iloc":
        result = self._handle_lowerdim_multi_index_axis0(tup)
        if result is not None:
            return result

    if len(tup) > self.ndim:
        raise IndexingError("Too many indexers. handle elsewhere")

    for i, key in enumerate(tup):
        if is_label_like(key):
            # We don't need to check for tuples here because those are
            # caught by the _is_nested_tuple_indexer check above.
            section = self._getitem_axis(key, axis=i)

            # We should never have a scalar section here, because
            # _getitem_lowerdim is only called after a check for
            # is_scalar_access, which that would be.
            if section.ndim == self.ndim:
                # we're in the middle of slicing through a MultiIndex
                # revise the key wrt to `section` by inserting an _NS
                new_key = tup[:i] + (_NS,) + tup[i + 1 :]

        else:
            # Note: the section.ndim == self.ndim check above
            # rules out having DataFrame here, so we dont need to worry
            # about transposing.
            new_key = tup[:i] + tup[i + 1 :]

            if len(new_key) == 1:
                new_key = new_key[0]

        # Slices should return views, but calling iloc/loc with a null
        # slice returns a new object.
        if com.is_null_slice(new_key):
            return section
        # This is an elided recursive call to iloc/loc
        return getattr(section, self.name)[new_key]

    raise IndexingError("not applicable")

def _getitem_nested_tuple(self, tup: Tuple):
    # we have a nested tuple so have at least 1 multi-index level
    # we should be able to match up the dimensionality here

```

```

# we have too many indexers for our dim, but have at least 1
# multi-index dimension, try to see if we have something like
# a tuple passed to a series with a multi-index
if len(tup) > self.ndim:
    if self.name != "loc":
        # This should never be reached, but lets be explicit about it
        raise ValueError("Too many indices")
    result = self._handle_lowerdim_multi_index_axis0(tup)
    if result is not None:
        return result

    # this is a series with a multi-index specified a tuple of
    # selectors
    axis = self.axis or 0
    return self._getitem_axis(tup, axis=axis)

# handle the multi-axis by taking sections and reducing
# this is iterative
obj = self.obj
axis = 0
for key in tup:

    if com.is_null_slice(key):
        axis += 1
        continue

    current_ndim = obj.ndim
    obj = getattr(obj, self.name)._getitem_axis(key, axis=axis)
    axis += 1

    # if we have a scalar, we are done
    if is_scalar(obj) or not hasattr(obj, "ndim"):
        break

    # has the dim of the obj changed?
    # GH 7199
    if obj.ndim < current_ndim:
        axis -= 1

return obj

def _convert_to_indexer(self, key, axis: int, is_setter: bool = False):
    raise AbstractMethodError(self)

def __getitem__(self, key):
    if type(key) is tuple:
        key = tuple(com.apply_if_callable(x, self.obj) for x in key)
        if self._is_scalar_access(key):
            try:
                return self.obj._get_value(*key, takeable=self._takeable)
            except (KeyError, IndexError, AttributeError):
                # AttributeError for IntervalTree get_value
                pass
        return self._getitem_tuple(key)
    else:
        # we by definition only have the 0th axis
        axis = self.axis or 0

        maybe_callable = com.apply_if_callable(key, self.obj)
        return self._getitem_axis(maybe_callable, axis=axis)

def _is_scalar_access(self, key: Tuple):
    raise NotImplementedError()

```

```

def __getitem_tuple(self, tup: Tuple):
    raise AbstractMethodError(self)

def __getitem_axis(self, key, axis: int):
    raise NotImplementedError()

def __has_valid_setitem_indexer(self, indexer) -> bool:
    raise AbstractMethodError(self)

def __getbool_axis(self, key, axis: int):
    # caller is responsible for ensuring non-None axis
    labels = self.obj._get_axis(axis)
    key = check_bool_indexer(labels, key)
    inds = key.nonzero()[0]
    return self.obj._take_with_is_copy(inds, axis=axis)

@doc(IndexingMixin.loc)
class _LocIndexer(_LocationIndexer):
    _takeable: bool = False
    _valid_types = (
        "labels (MUST BE IN THE INDEX), slices of labels (BOTH "
        "endpoints included! Can be slices of integers if the "
        "index is integers), listlike of labels, boolean"
    )

# -----
# Key Checks

@doc(_LocationIndexer._validate_key)
def _validate_key(self, key, axis: int):

    # valid for a collection of labels (we check their presence later)
    # slice of labels (where start-end in labels)
    # slice of integers (only if in the labels)
    # boolean
    pass

def __has_valid_setitem_indexer(self, indexer) -> bool:
    return True

def __is_scalar_access(self, key: Tuple) -> bool:
    """
    Returns
    -----
    bool
    """
    # this is a shortcut accessor to both .loc and .iloc
    # that provide the equivalent access of .at and .iat
    # a) avoid getting things via sections and (to minimize dtype changes)
    # b) provide a performant path
    if len(key) != self.ndim:
        return False

    for i, k in enumerate(key):
        if not is_scalar(k):
            return False

        ax = self.obj.axes[i]
        if isinstance(ax, ABCMultiIndex):
            return False

```

```

        if isinstance(k, str) and ax._supports_partial_string_indexing:
            # partial string indexing, df.loc['2000', 'A']
            # should not be considered scalar
            return False

        if not ax.is_unique:
            return False

    return True

# -----
# MultiIndex Handling

def _multi_take_opportunity(self, tup: Tuple) -> bool:
    """
    Check whether there is the possibility to use ``_multi_take``.

    Currently the limit is that all axes being indexed, must be indexed with
    list-likes.

    Parameters
    -----
    tup : tuple
        Tuple of indexers, one per axis.

    Returns
    -----
    bool
        Whether the current indexing,
        can be passed through `_multi_take`.
    """
    if not all(is_list_like_indexer(x) for x in tup):
        return False

    # just too complicated
    if any(com.is_bool_indexer(x) for x in tup):
        return False

    return True

def _multi_take(self, tup: Tuple):
    """
    Create the indexers for the passed tuple of keys, and
    executes the take operation. This allows the take operation to be
    executed all at once, rather than once for each dimension.
    Improving efficiency.

    Parameters
    -----
    tup : tuple
        Tuple of indexers, one per axis.

    Returns
    -----
    values: same type as the object being indexed
    """
    # GH 836
    d = {
        axis: self._get_listlike_indexer(key, axis)
        for (key, axis) in zip(tup, self.obj._AXIS_ORDERS)
    }
    return self.obj._reindex_with_indexers(d, copy=True, allow_dups=True)

```

```

# -----
def __getitem_iterable(self, key, axis: int):
    """
    Index current object with an an iterable collection of keys.

    Parameters
    -----
    key : iterable
        Targeted labels.
    axis: int
        Dimension on which the indexing is being made.

    Raises
    -----
    KeyError
        If no key was found. Will change in the future to raise if not all
        keys were found.

    Returns
    -----
    scalar, DataFrame, or Series: indexed value(s).
    """
    # we assume that not com.is_bool_indexer(key), as that is
    # handled before we get here.
    self._validate_key(key, axis)

    # A collection of keys
    keyarr, indexer = self._get_listlike_indexer(key, axis, raise_missing=False)
    return self.obj._reindex_with_indexers(
        {axis: [keyarr, indexer]}, copy=True, allow_dups=True
    )

def __getitem_tuple(self, tup: Tuple):
    try:
        return self.__getitem_lowerdim(tup)
    except IndexingError:
        pass

    # no multi-index, so validate all of the indexers
    self._has_valid_tuple(tup)

    # ugly hack for GH #836
    if self._multi_take_opportunity(tup):
        return self._multi_take(tup)

    return self.__getitem_tuple_same_dim(tup)

def __get_label(self, label, axis: int):
    # GH#5667 this will fail if the label is not present in the axis.
    return self.obj.xs(label, axis=axis)

def __handle_lowerdim_multi_index_axis0(self, tup: Tuple):
    # we have an axis0 multi-index, handle or raise
    axis = self.axis or 0
    try:
        # fast path for series or for tup devoid of slices
        return self.__get_label(tup, axis=axis)
    except TypeError:
        # slices are unhashable
        pass
    except KeyError as ek:
        # raise KeyError if number of indexers match

```

```

        # else IndexError will be raised
        if len(tup) <= self.obj.index.nlevels and len(tup) > self.ndim:
            raise ek

    return None

def _getitem_axis(self, key, axis: int):
    key = item_from_zerodim(key)
    if is_iterator(key):
        key = list(key)

    labels = self.obj._get_axis(axis)
    key = labels._get_partial_string_timestamp_match_key(key)

    if isinstance(key, slice):
        self._validate_key(key, axis)
        return self._get_slice_axis(key, axis=axis)
    elif com.is_bool_indexer(key):
        return self._getbool_axis(key, axis=axis)
    elif is_list_like_indexer(key):

        # convert various list-like indexers
        # to a list of keys
        # we will use the *values* of the object
        # and NOT the index if its a PandasObject
        if isinstance(labels, ABCMultiIndex):

            if isinstance(key, (ABCSeries, np.ndarray)) and key.ndim <= 1:
                # Series, or 0,1 ndim ndarray
                # GH 14730
                key = list(key)
            elif isinstance(key, ABCDataFrame):
                # GH 15438
                raise NotImplementedError(
                    "Indexing a MultiIndex with a "
                    "DataFrame key is not "
                    "implemented"
                )
            elif hasattr(key, "ndim") and key.ndim > 1:
                raise NotImplementedError(
                    "Indexing a MultiIndex with a "
                    "multidimensional key is not "
                    "implemented"
                )

            if (
                not isinstance(key, tuple)
                and len(key)
                and not isinstance(key[0], tuple)
            ):
                key = tuple([key])

        # an iterable multi-selection
        if not (isinstance(key, tuple) and isinstance(labels, ABCMultiIndex)):

            if hasattr(key, "ndim") and key.ndim > 1:
                raise ValueError("Cannot index with multidimensional key")

            return self._getitem_iterable(key, axis=axis)

        # nested tuple slicing
        if is_nested_tuple(key, labels):
            locs = labels.get_locs(key)

```

```

        indexer = [slice(None)] * self.ndim
        indexer[axis] = locs
        return self.obj.iloc[tuple(indexer)]

    # fall thru to straight lookup
    self._validate_key(key, axis)
    return self._get_label(key, axis=axis)

def _get_slice_axis(self, slice_obj: slice, axis: int):
    """
    This is pretty simple as we just have to deal with labels.
    """
    # caller is responsible for ensuring non-None axis
    obj = self.obj
    if not need_slice(slice_obj):
        return obj.copy(deep=False)

    labels = obj._get_axis(axis)
    indexer = labels.slice_indexer(
        slice_obj.start, slice_obj.stop, slice_obj.step, kind="loc"
    )

    if isinstance(indexer, slice):
        return self.obj._slice(indexer, axis=axis)
    else:
        # DatetimeIndex overrides Index.slice_indexer and may
        # return a DatetimeIndex instead of a slice object.
        return self.obj.take(indexer, axis=axis)

def _convert_to_indexer(self, key, axis: int, is_setter: bool = False):
    """
    Convert indexing key into something we can use to do actual fancy
    indexing on a ndarray.

    Examples
    ix[:5] -> slice(0, 5)
    ix[[1,2,3]] -> [1,2,3]
    ix[['foo', 'bar', 'baz']] -> [i, j, k] (indices of foo, bar, baz)

    Going by Zen of Python?
    'In the face of ambiguity, refuse the temptation to guess.'
    raise AmbiguousIndexError with integer labels?
    - No, prefer label-based indexing
    """
    labels = self.obj._get_axis(axis)

    if isinstance(key, slice):
        return labels._convert_slice_indexer(key, kind="loc")

    # see if we are positional in nature
    is_int_index = labels.is_integer()
    is_int_positional = is_integer(key) and not is_int_index

    if is_scalar(key) or isinstance(labels, ABCMultiIndex):
        # Otherwise get_loc will raise InvalidIndexError

        # if we are a label return me
        try:
            return labels.get_loc(key)
        except LookupError:
            if isinstance(key, tuple) and isinstance(labels, ABCMultiIndex):
                if len(key) == labels.nlevels:
                    return {"key": key}

```

```

        raise
    except TypeError:
        pass
    except ValueError:
        if not is_int_positional:
            raise

    # a positional
    if is_int_positional:

        # if we are setting and its not a valid location
        # its an insert which fails by definition

        # always valid
        return {"key": key}

    if is_nested_tuple(key, labels):
        return labels.get_locs(key)

    elif is_list_like_indexer(key):

        if com.is_bool_indexer(key):
            key = check_bool_indexer(labels, key)
            (inds,) = key.nonzero()
            return inds
        else:
            # When setting, missing keys are not allowed, even with .loc:
            return self._get_listlike_indexer(key, axis, raise_missing=True)[1]
    else:
        try:
            return labels.get_loc(key)
        except LookupError:
            # allow a not found key only if we are a setter
            if not is_list_like_indexer(key):
                return {"key": key}
            raise

def _get_listlike_indexer(self, key, axis: int, raise_missing: bool = False):
    """
    Transform a list-like of keys into a new index and an indexer.

    Parameters
    -----
    key : list-like
        Targeted labels.
    axis: int
        Dimension on which the indexing is being made.
    raise_missing: bool, default False
        Whether to raise a KeyError if some labels were not found.
        Will be removed in the future, and then this method will always behave as
        if ``raise_missing=True``.

    Raises
    -----
    KeyError
        If at least one key was requested but none was found, and
        raise_missing=True.

    Returns
    -----
    keyarr: Index
        New index (coinciding with 'key' if the axis is unique).
    values : array-like
    """

```

```

        Indexer for the return object, -1 denotes keys not found.
"""
ax = self.obj._get_axis(axis)

# Have the index compute an indexer or return None
# if it cannot handle:
indexer, keyarr = ax._convert_listlike_indexer(key)
# We only act on all found values:
if indexer is not None and (indexer != -1).all():
    self._validate_read_indexer(key, indexer, axis, raise_missing=raise_missing)
    return ax[indexer], indexer

if ax.is_unique and not getattr(ax, "is_overlapping", False):
    indexer = ax.get_indexer_for(key)
    keyarr = ax.reindex(keyarr)[0]
else:
    keyarr, indexer, new_indexer = ax._reindex_non_unique(keyarr)

self._validate_read_indexer(keyarr, indexer, axis, raise_missing=raise_missing)
return keyarr, indexer

def _validate_read_indexer(
    self, key, indexer, axis: int, raise_missing: bool = False
):
    """
    Check that indexer can be used to return a result.

    e.g. at least one element was found,
    unless the list of keys was actually empty.

    Parameters
    -----
    key : list-like
        Targeted labels (only used to show correct error message).
    indexer: array-like of booleans
        Indices corresponding to the key,
        (with -1 indicating not found).
    axis: int
        Dimension on which the indexing is being made.
    raise_missing: bool
        Whether to raise a KeyError if some labels are not found. Will be
        removed in the future, and then this method will always behave as
        if raise_missing=True.

    Raises
    -----
    KeyError
        If at least one key was requested but none was found, and
        raise_missing=True.
"""
    ax = self.obj._get_axis(axis)

    if len(key) == 0:
        return

    # Count missing values:
    missing = (indexer < 0).sum()

    if missing:
        if missing == len(indexer):
            axis_name = self.obj._get_axis_name(axis)
            raise KeyError(f"None of [{key}] are in the [{axis_name}]")

```

```

# We (temporarily) allow for some missing keys with .loc, except in
# some cases (e.g. setting) in which "raise_missing" will be False
if raise_missing:
    not_found = list(set(key) - set(ax))
    raise KeyError(f"{not_found} not in index")

# we skip the warning on Categorical
# as this check is actually done (check for
# non-missing values), but a bit later in the
# code, so we want to avoid warning & then
# just raising
if not ax.is_categorical():
    raise KeyError(
        "Passing list-likes to .loc or [] with any missing labels "
        "is no longer supported, see "
        "https://pandas.pydata.org/pandas-docs/stable/user\_guide/indexing.html#deprecate-loc-reindex-listlike" # noqa:E501
    )

@doc(IndexingMixin.iloc)
class _iLocIndexer(_LocationIndexer):
    _valid_types = (
        "integer, integer slice (START point is INCLUDED, END "
        "point is EXCLUDED), listlike of integers, boolean array"
    )
    _takeable = True

    # -----
    # Key Checks

    def _validate_key(self, key, axis: int):
        if com.is_bool_indexer(key):
            if hasattr(key, "index") and isinstance(key.index, Index):
                if key.index.inferred_type == "integer":
                    raise NotImplementedError(
                        "iLocation based boolean "
                        "indexing on an integer type "
                        "is not available"
                    )
                raise ValueError(
                    "iLocation based boolean indexing cannot use "
                    "an indexable as a mask"
                )
        return

        if isinstance(key, slice):
            return
        elif is_integer(key):
            self._validate_integer(key, axis)
        elif isinstance(key, tuple):
            # a tuple should already have been caught by this point
            # so don't treat a tuple as a valid indexer
            raise IndexingError("Too many indexers")
        elif is_list_like_indexer(key):
            arr = np.array(key)
            len_axis = len(self.obj._get_axis(axis))

            # check that the key has a numeric dtype
            if not is_numeric_dtype(arr.dtype):
                raise IndexError(f".iloc requires numeric indexers, got {arr}")

            # check that the key does not exceed the maximum size of the index

```

```

        if len(arr) and (arr.max() >= len_axis or arr.min() < -len_axis):
            raise IndexError("positional indexers are out-of-bounds")
    else:
        raise ValueError(f"Can only index by location with a [{self._valid_types}]")

def _has_valid_setitem_indexer(self, indexer) -> bool:
    """
    Validate that a positional indexer cannot enlarge its target
    will raise if needed, does not modify the indexer externally.

    Returns
    ------
    bool
    """
    if isinstance(indexer, dict):
        raise IndexError("iloc cannot enlarge its target object")
    else:
        if not isinstance(indexer, tuple):
            indexer = _tuplify(self.ndim, indexer)
        for ax, i in zip(self.obj.axes, indexer):
            if isinstance(i, slice):
                # should check the stop slice?
                pass
            elif is_list_like_indexer(i):
                # should check the elements?
                pass
            elif is_integer(i):
                if i >= len(ax):
                    raise IndexError("iloc cannot enlarge its target object")
            elif isinstance(i, dict):
                raise IndexError("iloc cannot enlarge its target object")

    return True

def _is_scalar_access(self, key: Tuple) -> bool:
    """
    Returns
    ------
    bool
    """
    # this is a shortcut accessor to both .loc and .iloc
    # that provide the equivalent access of .at and .iat
    # a) avoid getting things via sections and (to minimize dtype changes)
    # b) provide a performant path
    if len(key) != self.ndim:
        return False

    for k in key:
        if not is_integer(k):
            return False

    return True

def _validate_integer(self, key: int, axis: int) -> None:
    """
    Check that 'key' is a valid position in the desired axis.

    Parameters
    ------
    key : int
        Requested position.
    axis : int
        Desired axis.
    """

```

```
Raises
-----
IndexError
    If 'key' is not a valid position in axis 'axis'.
"""

len_axis = len(self.obj._get_axis(axis))
if key >= len_axis or key < -len_axis:
    raise IndexError("single positional indexer is out-of-bounds")

# -----
def _getitem_tuple(self, tup: Tuple):
    self._has_valid_tuple(tup)
    try:
        return self._getitem_lowerdim(tup)
    except IndexingError:
        pass

    return self._getitem_tuple_same_dim(tup)

def _get_list_axis(self, key, axis: int):
    """
    Return Series values by list or array of integers.

    Parameters
    -----
    key : list-like positional indexer
    axis : int

    Returns
    -----
    Series object

    Notes
    -----
    `axis` can only be zero.
    """
    try:
        return self.obj._take_with_is_copy(key, axis=axis)
    except IndexError as err:
        # re-raise with different error message
        raise IndexError("positional indexers are out-of-bounds") from err

def _getitem_axis(self, key, axis: int):
    if isinstance(key, slice):
        return self._get_slice_axis(key, axis=axis)

    if isinstance(key, list):
        key = np.asarray(key)

    if com.is_bool_indexer(key):
        self._validate_key(key, axis)
        return self._getbool_axis(key, axis=axis)

    # a list of integers
    elif is_list_like_indexer(key):
        return self._get_list_axis(key, axis=axis)

    # a single integer
    else:
        key = item_from_zerodim(key)
```

```

        if not is_integer(key):
            raise TypeError("Cannot index by location index with a non-integer key")

        # validate the location
        self._validate_integer(key, axis)

    return self.obj._ixs(key, axis=axis)

def _get_slice_axis(self, slice_obj: slice, axis: int):
    # caller is responsible for ensuring non-None axis
    obj = self.obj

    if not need_slice(slice_obj):
        return obj.copy(deep=False)

    labels = obj._get_axis(axis)
    labels._validate_positional_slice(slice_obj)
    return self.obj._slice(slice_obj, axis=axis)

def _convert_to_indexer(self, key, axis: int, is_setter: bool = False):
    """
    Much simpler as we only have to deal with our valid types.
    """
    return key

def _get_setitem_indexer(self, key):
    # GH#32257 Fall through to let numpy do validation
    return key

# -----
def _setitem_with_indexer(self, indexer, value):
    """
    _setitem_with_indexer is for setting values on a Series/DataFrame
    using positional indexers.

    If the relevant keys are not present, the Series/DataFrame may be
    expanded.

    This method is currently broken when dealing with non-unique Indexes,
    since it goes from positional indexers back to labels when calling
    BlockManager methods, see GH#12991, GH#22046, GH#15686.
    """
    # also has the side effect of consolidating in-place
    from pandas import Series

    info_axis = self.obj._info_axis_number

    # maybe partial set
    take_split_path = self.obj._is_mixed_type

    # if there is only one block/type, still have to take split path
    # unless the block is one-dimensional or it can hold the value
    if not take_split_path and self.obj._mgr.blocks:
        (blk,) = self.obj._mgr.blocks
        if 1 < blk.ndim:  # in case of dict, keys are indices
            val = list(value.values()) if isinstance(value, dict) else value
            take_split_path = not blk._can_hold_element(val)

    # if we have any multi-indexes that have non-trivial slices
    # (not null slices) then we must take the split path, xref
    # GH 10360, GH 27841

```

```

if isinstance(indexer, tuple) and len(indexer) == len(self.obj.axes):
    for i, ax in zip(indexer, self.obj.axes):
        if isinstance(ax, ABCMultiIndex) and not (
            is_integer(i) or com.is_null_slice(i)
        ):
            take_split_path = True
            break

if isinstance(indexer, tuple):
    nindexer = []
    for i, idx in enumerate(indexer):
        if isinstance(idx, dict):

            # reindex the axis to the new value
            # and set inplace
            key, _ = convert_missing_indexer(idx)

            # if this is the items axes, then take the main missing
            # path first
            # this correctly sets the dtype and avoids cache issues
            # essentially this separates out the block that is needed
            # to possibly be modified
            if self.ndim > 1 and i == info_axis:

                # add the new item, and set the value
                # must have all defined axes if we have a scalar
                # or a list-like on the non-info axes if we have a
                # list-like
                len_non_info_axes = (
                    len(_ax) for _i, _ax in enumerate(self.obj.axes) if _i != i
                )
                if any(not l for l in len_non_info_axes):
                    if not is_list_like_indexer(value):
                        raise ValueError(
                            "cannot set a frame with no "
                            "defined index and a scalar"
                        )
                    self.obj[key] = value
                    return

                # add a new item with the dtype setup
                self.obj[key] = _infer_fill_value(value)

                new_indexer = convert_from_missing_indexer_tuple(
                    indexer, self.obj.axes
                )
                self._setitem_with_indexer(new_indexer, value)

            return

        # reindex the axis
        # make sure to clear the cache because we are
        # just replacing the block manager here
        # so the object is the same
        index = self.obj._get_axis(i)
        labels = index.insert(len(index), key)
        self.obj._mgr = self.obj.reindex(labels, axis=i)._mgr
        self.obj._maybe_update_cacher(clear=True)
        self.obj._is_copy = None

        nindexer.append(labels.get_loc(key))

else:

```

```

        nindexer.append(idx)

    indexer = tuple(nindexer)
else:

    indexer, missing = convert_missing_indexer(indexer)

    if missing:
        self._setitem_with_indexer_missing(indexer, value)
        return

# set
item_labels = self.obj._get_axis(info_axis)

# align and set the values
if take_split_path:
    # Above we only set take_split_path to True for 2D cases
    assert self.ndim == 2
    assert info_axis == 1

    if not isinstance(indexer, tuple):
        indexer = _tuplify(self.ndim, indexer)

    if isinstance(value, ABCSeries):
        value = self._align_series(indexer, value)

    info_idx = indexer[info_axis]
    if is_integer(info_idx):
        info_idx = [info_idx]
    labels = item_labels[info_idx]

    # Ensure we have something we can iterate over
    ilocs = info_idx
    if isinstance(info_idx, slice):
        ri = Index(range(len(self.obj.columns)))
        ilocs = ri[info_idx]

    plane_indexer = indexer[:1]
    lplane_indexer = length_of_indexer(plane_indexer[0], self.obj.index)
    # lplane_indexer gives the expected length of obj[indexer[0]]

    if len(labels) == 1:
        # We can operate on a single column

        # require that we are setting the right number of values that
        # we are indexing
        if is_list_like_indexer(value) and 0 != lplane_indexer != len(value):
            # Exclude zero-len for e.g. boolean masking that is all-false
            raise ValueError(
                "cannot set using a multi-index "
                "selection indexer with a different "
                "length than the value"
            )

    pi = plane_indexer[0] if lplane_indexer == 1 else plane_indexer

def isetter(loc, v):
    # positional setting on column loc
    ser = self.obj._ixs(loc, axis=1)

    # perform the equivalent of a setitem on the info axis
    # as we have a null slice or a slice with full bounds
    # which means essentially reassign to the columns of a

```

```

# multi-dim object
# GH6149 (null slice), GH10408 (full bounds)
if isinstance(pi, tuple) and all(
    com.is_null_slice(idx) or com.is_full_slice(idx, len(self.obj))
    for idx in pi
):
    ser = v
else:
    # set the item, possibly having a dtype change
    ser._consolidate_inplace()
    ser = ser.copy()
    ser._mgr = ser._mgr.setitem(indexer=pi, value=v)
    ser._maybe_update_cacher(clear=True)

# reset the sliced object if unique
self.obj._iset_item(loc, ser)

# we need an iterable, with a ndim of at least 1
# eg. don't pass through np.array(0)
if is_list_like_indexer(value) and getattr(value, "ndim", 1) > 0:

    # we have an equal len Frame
    if isinstance(value, ABCDataFrame):
        sub_indexer = list(indexer)
        multiindex_indexer = isinstance(labels, ABCMultiIndex)
        # TODO: we are implicitly assuming value.columns is unique

        for loc in ilocs:
            item = item_labels[loc]
            if item in value:
                sub_indexer[info_axis] = item
                v = self._align_series(
                    tuple(sub_indexer), value[item], multiindex_indexer
                )
            else:
                v = np.nan

            isetter(loc, v)

    # we have an equal len ndarray/convertible to our labels
    # hasattr first, to avoid coercing to ndarray without reason.
    # But we may be relying on the ndarray coercion to check ndim.
    # Why not just convert to an ndarray earlier on if needed?
    elif np.ndim(value) == 2:

        # note that this coerces the dtype if we are mixed
        # GH 7551
        value = np.array(value, dtype=object)
        if len(ilocs) != value.shape[1]:
            raise ValueError(
                "Must have equal len keys and value "
                "when setting with an ndarray"
            )

        for i, loc in enumerate(ilocs):
            # setting with a list, re-coerces
            isetter(loc, value[:, i].tolist())

    elif (
        len(labels) == 1
        and lplane_indexer == len(value)
        and not is_scalar(plane_indexer[0])
    ):

```

```

        # we have an equal len list/ndarray
        # We only get here with len(labels) == len(ilocs) == 1
        isetter(ilocs[0], value)

    elif lplane_indexer == 0 and len(value) == len(self.obj.index):
        # We get here in one case via .loc with a all-False mask
        pass

    else:
        # per-label values
        if len(ilocs) != len(value):
            raise ValueError(
                "Must have equal len keys and value "
                "when setting with an iterable"
            )

        for loc, v in zip(ilocs, value):
            isetter(loc, v)
    else:

        # scalar value
        for loc in ilocs:
            isetter(loc, value)

    else:
        if isinstance(indexer, tuple):

            # if we are setting on the info axis ONLY
            # set using those methods to avoid block-splitting
            # logic here
            if (
                len(indexer) > info_axis
                and is_integer(indexer[info_axis])
                and all(
                    com.is_null_slice(idx)
                    for i, idx in enumerate(indexer)
                    if i != info_axis
                )
                and item_labels.is_unique
            ):
                self.obj[item_labels[indexer[info_axis]]] = value
                return

        indexer = maybe_convert_ix(*indexer)

    if isinstance(value, (ABCSeries, dict)):
        # TODO(EA): ExtensionBlock.setitem this causes issues with
        # setting for extensionarrays that store dicts. Need to decide
        # if it's worth supporting that.
        value = self._align_series(indexer, Series(value))

    elif isinstance(value, ABCDataFrame):
        value = self._align_frame(indexer, value)

    # check for chained assignment
    self.obj._check_is_chained_assignment_possible()

    # actually do the set
    self.obj._consolidate_inplace()
    self.obj._mgr = self.obj._mgr.setitem(indexer=indexer, value=value)
    self.obj._maybe_update_cacher(clear=True)

def _setitem_with_indexer_missing(self, indexer, value):

```

```

"""
Insert new row(s) or column(s) into the Series or DataFrame.
"""
from pandas import Series

# reindex the axis to the new value
# and set inplace
if self.ndim == 1:
    index = self.obj.index
    new_index = index.insert(len(index), indexer)

    # we have a coerced indexer, e.g. a float
    # that matches in an Int64Index, so
    # we will not create a duplicate index, rather
    # index to that element
    # e.g. 0.0 -> 0
    # GH#12246
    if index.is_unique:
        new_indexer = index.get_indexer([new_index[-1]])
        if (new_indexer != -1).any():
            return self._setitem_with_indexer(new_indexer, value)

    # this preserves dtype of the value
    new_values = Series([value])._values
    if len(self.obj._values):
        # GH#22717 handle casting compatibility that np.concatenate
        # does incorrectly
        new_values = concat_compat([self.obj._values, new_values])
    self.obj._mgr = self.obj._constructor(
        new_values, index=new_index, name=self.obj.name
    )._mgr
    self.obj._maybe_update_cacher(clear=True)

elif self.ndim == 2:

    if not len(self.obj.columns):
        # no columns and scalar
        raise ValueError("cannot set a frame with no defined columns")

    if isinstance(value, ABCSeries):
        # append a Series
        value = value.reindex(index=self.obj.columns, copy=True)
        value.name = indexer

    else:
        # a list-list
        if is_list_like_indexer(value):
            # must have conforming columns
            if len(value) != len(self.obj.columns):
                raise ValueError("cannot set a row with mismatched columns")

            value = Series(value, index=self.obj.columns, name=indexer)

        self.obj._mgr = self.obj.append(value)._mgr
        self.obj._maybe_update_cacher(clear=True)

def _align_series(self, indexer, ser: ABCSeries, multiindex_indexer: bool = False):
    """
    Parameters
    -----
    indexer : tuple, slice, scalar
        Indexer used to get the locations that will be set to `ser`.
    ser : pd.Series

```

```

    Values to assign to the locations specified by `indexer`.
multiindex_indexer : boolean, optional
    Defaults to False. Should be set to True if `indexer` was from
    a `pd.MultiIndex`, to avoid unnecessary broadcasting.

Returns
-----
`np.array` of `ser` broadcast to the appropriate shape for assignment
to the locations selected by `indexer`
"""
if isinstance(indexer, (slice, np.ndarray, list, Index)):
    indexer = tuple([indexer])

if isinstance(indexer, tuple):

    # flatten np.ndarray indexers
    def ravel(i):
        return i.ravel() if isinstance(i, np.ndarray) else i

    indexer = tuple(map(ravel, indexer))

    aligners = [not com.is_null_slice(idx) for idx in indexer]
    sum_aligners = sum(aligners)
    single_aligner = sum_aligners == 1
    is_frame = self.ndim == 2
    obj = self.obj

    # are we a single alignable value on a non-primary
    # dim (e.g. panel: 1,2, or frame: 0) ?
    # hence need to align to a single axis dimension
    # rather than find all valid dims

    # frame
    if is_frame:
        single_aligner = single_aligner and aligners[0]

    # we have a frame, with multiple indexers on both axes; and a
    # series, so need to broadcast (see GH5206)
    if sum_aligners == self.ndim and all(is_sequence(_) for _ in indexer):
        ser = ser.reindex(obj.axes[0][indexer[0]], copy=True)._values

        # single indexer
        if len(indexer) > 1 and not multiindex_indexer:
            len_indexer = len(indexer[1])
            ser = np.tile(ser, len_indexer).reshape(len_indexer, -1).T

    return ser

for i, idx in enumerate(indexer):
    ax = obj.axes[i]

    # multiple aligners (or null slices)
    if is_sequence(idx) or isinstance(idx, slice):
        if single_aligner and com.is_null_slice(idx):
            continue
        new_ix = ax[idx]
        if not is_list_like_indexer(new_ix):
            new_ix = Index([new_ix])
        else:
            new_ix = Index(new_ix)
        if ser.index.equals(new_ix) or not len(new_ix):
            return ser._values.copy()

```

```

        return ser.reindex(new_ix)._values

    # 2 dims
    elif single_aligner:

        # reindex along index
        ax = self.obj.axes[1]
        if ser.index.equals(ax) or not len(ax):
            return ser._values.copy()
        return ser.reindex(ax)._values

    elif is_scalar(indexer):
        ax = self.obj._get_axis(1)

        if ser.index.equals(ax):
            return ser._values.copy()

        return ser.reindex(ax)._values

    raise ValueError("Incompatible indexer with Series")

def _align_frame(self, indexer, df: ABCDataFrame):
    is_frame = self.ndim == 2

    if isinstance(indexer, tuple):

        idx, cols = None, None
        sindexers = []
        for i, ix in enumerate(indexer):
            ax = self.obj.axes[i]
            if is_sequence(ix) or isinstance(ix, slice):
                if isinstance(ix, np.ndarray):
                    ix = ix.ravel()
                if idx is None:
                    idx = ax[ix]
                elif cols is None:
                    cols = ax[ix]
                else:
                    break
            else:
                sindexers.append(i)

        if idx is not None and cols is not None:

            if df.index.equals(idx) and df.columns.equals(cols):
                val = df.copy()._values
            else:
                val = df.reindex(idx, columns=cols)._values
            return val

    elif (isinstance(indexer, slice) or is_list_like_indexer(indexer)) and is_frame:
        ax = self.obj.index[indexer]
        if df.index.equals(ax):
            val = df.copy()._values
        else:

            # we have a multi-index and are trying to align
            # with a particular, level GH3738
            if (
                isinstance(ax, ABCMultiIndex)
                and isinstance(df.index, ABCMultiIndex)
                and ax.nlevels != df.index.nlevels
            ):

```

```

        raise TypeError(
            "cannot align on a multi-index with out "
            "specifying the join levels"
        )

        val = df.reindex(index=ax)._values
        return val

    raise ValueError("Incompatible indexer with DataFrame")

class _ScalarAccessIndexer(_NDFrameIndexerBase):
    """
    Access scalars quickly.
    """

    def __convert_key(self, key, is_setter: bool = False):
        raise AbstractMethodError(self)

    def __getitem__(self, key):
        if not isinstance(key, tuple):

            # we could have a convertible item here (e.g. Timestamp)
            if not is_list_like_indexer(key):
                key = tuple([key])
            else:
                raise ValueError("Invalid call for scalar access (getting)!")

        key = self.__convert_key(key)
        return self.obj._get_value(*key, takeable=self._takeable)

    def __setitem__(self, key, value):
        if isinstance(key, tuple):
            key = tuple(com.apply_if_callable(x, self.obj) for x in key)
        else:
            # scalar callable may return tuple
            key = com.apply_if_callable(key, self.obj)

        if not isinstance(key, tuple):
            key = _tuplify(self.ndim, key)
        if len(key) != self.ndim:
            raise ValueError("Not enough indexers for scalar access (setting)!")

        key = list(self.__convert_key(key, is_setter=True))
        self.obj._set_value(*key, value=value, takeable=self._takeable)

@doc(IndexingMixin.at)
class _AtIndexer(_ScalarAccessIndexer):
    _takeable = False

    def __convert_key(self, key, is_setter: bool = False):
        """
        Require they keys to be the same type as the index. (so we don't
        fallback)
        """
        # allow arbitrary setting
        if is_setter:
            return list(key)

        return key

    @property

```

```

def _axes_are_unique(self) -> bool:
    # Only relevant for self.ndim == 2
    assert self.ndim == 2
    return self.obj.index.is_unique and self.obj.columns.is_unique

def __getitem__(self, key):

    if self.ndim == 2 and not self._axes_are_unique:
        # GH#33041 fall back to .loc
        if not isinstance(key, tuple) or not all(is_scalar(x) for x in key):
            raise ValueError("Invalid call for scalar access (getting)!")
        return self.obj.loc[key]

    return super().__getitem__(key)

def __setitem__(self, key, value):
    if self.ndim == 2 and not self._axes_are_unique:
        # GH#33041 fall back to .loc
        if not isinstance(key, tuple) or not all(is_scalar(x) for x in key):
            raise ValueError("Invalid call for scalar access (setting)!")

        self.obj.loc[key] = value
    return

return super().__setitem__(key, value)

@doc(IndexingMixin.iat)
class _iAtIndexer(_ScalarAccessIndexer):
    _takeable = True

    def _convert_key(self, key, is_setter: bool = False):
        """
        Require integer args. (and convert to label arguments)
        """
        for a, i in zip(self.obj.axes, key):
            if not is_integer(i):
                raise ValueError("iAt based indexing can only have integer indexers")
        return key

def _tuplify(ndim: int, loc: Hashable) -> Tuple[Union[Hashable, slice], ...]:
    """
    Given an indexer for the first dimension, create an equivalent tuple
    for indexing over all dimensions.
    """

    Parameters
    -----
    ndim : int
    loc : object

    Returns
    -----
    tuple
    """
    _tup: List[Union[Hashable, slice]] = [slice(None, None) for _ in range(ndim)]
    _tup[0] = loc
    return tuple(_tup)

def convert_to_index_sliceable(obj: "DataFrame", key):
    """

```

```

If we are index sliceable, then return my slicer, otherwise return None.
"""
idx = obj.index
if isinstance(key, slice):
    return idx._convert_slice_indexer(key, kind="getitem")

elif isinstance(key, str):

    # we are an actual column
    if key in obj.columns:
        return None

    # We might have a datetimelike string that we can translate to a
    # slice here via partial string indexing
    if idx._supports_partial_string_indexing:
        try:
            return idx._get_string_slice(key)
        except (KeyError, ValueError, NotImplementedError):
            return None

return None

def check_bool_indexer(index: Index, key) -> np.ndarray:
    """
    Check if key is a valid boolean indexer for an object with such index and
    perform reindexing or conversion if needed.

    This function assumes that is_bool_indexer(key) == True.

    Parameters
    -----
    index : Index
        Index of the object on which the indexing is done.
    key : list-like
        Boolean indexer to check.

    Returns
    -----
    np.array
        Resulting key.

    Raises
    -----
    IndexError
        If the key does not have the same length as index.
    IndexingError
        If the index of the key is unalignable to index.
    """
    result = key
    if isinstance(key, ABCSeries) and not key.index.equals(index):
        result = result.reindex(index)
        mask = isna(result._values)
        if mask.any():
            raise IndexingError(
                "Unalignable boolean Series provided as "
                "indexer (index of the boolean Series and of "
                "the indexed object do not match)."
            )
        result = result.astype(bool)._values
    elif is_object_dtype(key):
        # key might be object-dtype bool, check_array_indexer needs bool array
        result = np.asarray(result, dtype=bool)

```

```

        result = check_array_indexer(index, result)
    else:
        result = check_array_indexer(index, result)

    return result

def convert_missing_indexer(indexer):
    """
    Reverse convert a missing indexer, which is a dict
    return the scalar indexer and a boolean indicating if we converted
    """
    if isinstance(indexer, dict):

        # a missing key (but not a tuple indexer)
        indexer = indexer["key"]

        if isinstance(indexer, bool):
            raise KeyError("cannot use a single bool to index into setitem")
        return indexer, True

    return indexer, False

def convert_from_missing_indexer_tuple(indexer, axes):
    """
    Create a filtered indexer that doesn't have any missing indexers.
    """
    def get_indexer(_i, _idx):
        return axes[_i].get_loc(_idx["key"]) if isinstance(_idx, dict) else _idx

    return tuple(get_indexer(_i, _idx) for _i, _idx in enumerate(indexer))

def maybe_convert_ix(*args):
    """
    We likely want to take the cross-product.
    """
    ixify = True
    for arg in args:
        if not isinstance(arg, (np.ndarray, list, ABCSeries, Index)):
            ixify = False

    if ixify:
        return np.ix_(*args)
    else:
        return args

def is_nested_tuple(tup, labels) -> bool:
    """
    Returns
    -----
    bool
    """
    # check for a compatible nested tuple and multiindexes among the axes
    if not isinstance(tup, tuple):
        return False

    for k in tup:
        if is_list_like(k) or isinstance(k, slice):
            return isinstance(labels, ABCMultiIndex)

```

```

    return False

def is_label_like(key) -> bool:
    """
    Returns
    -----
    bool
    """
    # select a label or row
    return not isinstance(key, slice) and not is_list_like_indexer(key)

def need_slice(obj) -> bool:
    """
    Returns
    -----
    bool
    """
    return (
        obj.start is not None
        or obj.stop is not None
        or (obj.step is not None and obj.step != 1)
    )

def _non_reducing_slice(slice_):
    """
    Ensuse that a slice doesn't reduce to a Series or Scalar.

    Any user-paseed `subset` should have this called on it
    to make sure we're always working with DataFrames.
    """
    # default to column slice, like DataFrame
    # ['A', 'B'] -> IndexSlices[:, ['A', 'B']]
    kinds = (ABCSeries, np.ndarray, Index, list, str)
    if isinstance(slice_, kinds):
        slice_ = IndexSlice[:, slice_]

def pred(part) -> bool:
    """
    Returns
    -----
    bool
        True if slice does *not* reduce,
        False if `part` is a tuple.
    """
    # true when slice does *not* reduce, False when part is a tuple,
    # i.e. MultiIndex slice
    return (isinstance(part, slice) or is_list_like(part)) and not isinstance(
        part, tuple
    )

if not is_list_like(slice_):
    if not isinstance(slice_, slice):
        # a 1-d slice, like df.loc[1]
        slice_ = [[slice_]]
    else:
        # slice(a, b, c)
        slice_ = [slice_]  # to tuplize later
else:
    slice_ = [part if pred(part) else [part] for part in slice_]

```

```
return tuple(slice_)

def _maybe_numeric_slice(df, slice_, include_bool=False):
    """
    Want nice defaults for background_gradient that don't break
    with non-numeric data. But if slice_ is passed go with that.
    """
    if slice_ is None:
        dtypes = [np.number]
        if include_bool:
            dtypes.append(bool)
        slice_ = IndexSlice[:, df.select_dtypes(include=dtypes).columns]
    return slice_
```

<https://github.com/numpy/numpy/blob/32f382c86f384a02eeb87f67fea806e6b9961b5c/numpy/core/fromnumeric.py>

```
"""Module containing non-deprecated functions borrowed from Numeric.

"""

import functools
import types
import warnings

import numpy as np
from . import multiarray as mu
from . import overrides
from . import umath as um
from . import numerictypes as nt
from ._asarray import asarray, array, asanyarray
from .multiarray import concatenate
from . import _methods

_dt_ = nt.sctype2char

# functions that are methods
_all__ = [
    'alen', 'all', 'alltrue', 'amax', 'amin', 'any', 'argmax',
    'argmin', 'argpartition', 'argsort', 'around', 'choose', 'clip',
    'compress', 'cumprod', 'cumproduct', 'cumsum', 'diagonal', 'mean',
    'ndim', 'nonzero', 'partition', 'prod', 'product', 'ptp', 'put',
    'ravel', 'repeat', 'reshape', 'resize', 'round_',
    'searchsorted', 'shape', 'size', 'sometrue', 'sort', 'squeeze',
    'std', 'sum', 'swapaxes', 'take', 'trace', 'transpose', 'var',
]
_gentype = types.GeneratorType
# save away Python sum
_sum_ = sum

array_function_dispatch = functools.partial(
    overrides.array_function_dispatch, module='numpy')

# functions that are now methods
def _wrapit(obj, method, *args, **kwds):
    try:
        wrap = obj.__array_wrap__
    except AttributeError:
        wrap = None
    result = getattr(asarray(obj), method)(*args, **kwds)
    if wrap:
        if not isinstance(result, mu.ndarray):
            result = asarray(result)
        result = wrap(result)
    return result

def _wrapfunc(obj, method, *args, **kwds):
    bound = getattr(obj, method, None)
    if bound is None:
        return _wrapit(obj, method, *args, **kwds)
```

```

try:
    return bound(*args, **kwds)
except TypeError:
    # A TypeError occurs if the object does have such a method in its
    # class, but its signature is not identical to that of NumPy's. This
    # situation has occurred in the case of a downstream library like
    # 'pandas'.
    #
    # Call _wrapit from within the except clause to ensure a potential
    # exception has a traceback chain.
    return _wrapit(obj, method, *args, **kwds)

def _wrapreduction(obj, ufunc, method, axis, dtype, out, **kwargs):
    passkwargs = {k: v for k, v in kwargs.items()
                  if v is not np._NoValue}

    if type(obj) is not mu.ndarray:
        try:
            reduction = getattr(obj, method)
        except AttributeError:
            pass
        else:
            # This branch is needed for reductions like any which don't
            # support a dtype.
            if dtype is not None:
                return reduction(axis=axis, dtype=dtype, out=out, **passkwargs)
            else:
                return reduction(axis=axis, out=out, **passkwargs)

    return ufunc.reduce(obj, axis, dtype, out, **passkwargs)

```

```

def _take_dispatcher(a, indices, axis=None, out=None, mode=None):
    return (a, out)

```

```

@array_function_dispatch(_take_dispatcher)
def take(a, indices, axis=None, out=None, mode='raise'):
    """
    Take elements from an array along an axis.

```

When axis is not None, this function does the same thing as "fancy" indexing (indexing arrays using arrays); however, it can be easier to use if you need elements along a given axis. A call such as ``np.take(arr, indices, axis=3)`` is equivalent to ``arr[:, :, :, indices, ...]``.

Explained without fancy indexing, this is equivalent to the following use of `ndindex`, which sets each of ``ii``, ``jj``, and ``kk`` to a tuple of indices::

```

Ni, Nk = a.shape[:axis], a.shape[axis+1:]
Nj = indices.shape
for ii in ndindex(Ni):
    for jj in ndindex(Nj):
        for kk in ndindex(Nk):
            out[ii + jj + kk] = a[ii + (indices[jj],) + kk]

```

Parameters

a : array_like (Ni..., M, Nk...)

```
The source array.  
indices : array_like (Nj...)  
    The indices of the values to extract.  
  
.. versionadded:: 1.8.0  
  
    Also allow scalars for indices.  
axis : int, optional  
    The axis over which to select values. By default, the flattened  
    input array is used.  
out : ndarray, optional (Ni..., Nj..., Nk...)  
    If provided, the result will be placed in this array. It should  
    be of the appropriate shape and dtype. Note that `out` is always  
    buffered if `mode='raise'`; use other modes for better performance.  
mode : {'raise', 'wrap', 'clip'}, optional  
    Specifies how out-of-bounds indices will behave.  
  
    * 'raise' -- raise an error (default)  
    * 'wrap' -- wrap around  
    * 'clip' -- clip to the range  
  
    'clip' mode means that all indices that are too large are replaced  
    by the index that addresses the last element along that axis. Note  
    that this disables indexing with negative numbers.
```

Returns

```
-----  
out : ndarray (Ni..., Nj..., Nk...)  
    The returned array has the same type as `a`.
```

See Also

```
-----  
compress : Take elements using a boolean mask  
ndarray.take : equivalent method  
take_along_axis : Take elements by matching the array and the index arrays
```

Notes

```
-----  
By eliminating the inner loop in the description above, and using `s_` to  
build simple slice objects, `take` can be expressed in terms of applying  
fancy indexing to each 1-d slice::
```

```
Ni, Nk = a.shape[:axis], a.shape[axis+1:]  
for ii in ndindex(Ni):  
    for kk in ndindex(Nj):  
        out[ii + s_[:, :, :] + kk] = a[ii + s_[:, :, :] + kk][indices]
```

```
For this reason, it is equivalent to (but faster than) the following use  
of `apply_along_axis`::
```

```
out = np.apply_along_axis(lambda a_1d: a_1d[indices], axis, a)
```

Examples

```
-----  
>>> a = [4, 3, 5, 7, 6, 8]  
>>> indices = [0, 1, 4]  
>>> np.take(a, indices)  
array([4, 3, 6])
```

In this example if `a` is an ndarray, "fancy" indexing can be used.

```
>>> a = np.array(a)
```

```
>>> a[indices]
array([4, 3, 6])

If `indices` is not one dimensional, the output also has these dimensions.

>>> np.take(a, [[0, 1], [2, 3]])
array([[4, 3],
       [5, 7]])
"""
return _wrapfunc(a, 'take', indices, axis=axis, out=out, mode=mode)

def _reshape_dispatcher(a, newshape, order=None):
    return (a,)

# not deprecated --- copy if necessary, view otherwise
@array_function_dispatch(_reshape_dispatcher)
def reshape(a, newshape, order='C'):
    """
    Gives a new shape to an array without changing its data.

    Parameters
    -----
    a : array_like
        Array to be reshaped.
    newshape : int or tuple of ints
        The new shape should be compatible with the original shape. If
        an integer, then the result will be a 1-D array of that length.
        One shape dimension can be -1. In this case, the value is
        inferred from the length of the array and remaining dimensions.
    order : {'C', 'F', 'A'}, optional
        Read the elements of `a` using this index order, and place the
        elements into the reshaped array using this index order. 'C'
        means to read / write the elements using C-like index order,
        with the last axis index changing fastest, back to the first
        axis index changing slowest. 'F' means to read / write the
        elements using Fortran-like index order, with the first index
        changing fastest, and the last index changing slowest. Note that
        the 'C' and 'F' options take no account of the memory layout of
        the underlying array, and only refer to the order of indexing.
        'A' means to read / write the elements in Fortran-like index
        order if `a` is Fortran *contiguous* in memory, C-like order
        otherwise.

    Returns
    -----
    reshaped_array : ndarray
        This will be a new view object if possible; otherwise, it will
        be a copy. Note there is no guarantee of the *memory layout* (C- or
        Fortran- contiguous) of the returned array.

    See Also
    -----
    ndarray.reshape : Equivalent method.

    Notes
    -----
    It is not always possible to change the shape of an array without
    copying the data. If you want an error to be raised when the data is copied,
    you should assign the new shape to the shape attribute of the array:::

    >>> a = np.zeros((10, 2))
```

```

# A transpose makes the array non-contiguous
>>> b = a.T

# Taking a view makes it possible to modify the shape without modifying
# the initial object.
>>> c = b.view()
>>> c.shape = (20)
Traceback (most recent call last):
...
AttributeError: Incompatible shape for in-place modification. Use
`reshape()` to make a copy with the desired shape.

```

The `order` keyword gives the index ordering both for *fetching* the values from `a`, and then *placing* the values into the output array.
For example, let's say you have an array:

```

>>> a = np.arange(6).reshape((3, 2))
>>> a
array([[0, 1],
       [2, 3],
       [4, 5]])

```

You can think of reshaping as first raveling the array (using the given index order), then inserting the elements from the raveled array into the new array using the same kind of index ordering as was used for the raveling.

```

>>> np.reshape(a, (2, 3)) # C-like index ordering
array([[0, 1, 2],
       [3, 4, 5]])
>>> np.reshape(np.ravel(a), (2, 3)) # equivalent to C ravel then C reshape
array([[0, 1, 2],
       [3, 4, 5]])
>>> np.reshape(a, (2, 3), order='F') # Fortran-like index ordering
array([[0, 4, 3],
       [2, 1, 5]])
>>> np.reshape(np.ravel(a, order='F'), (2, 3), order='F')
array([[0, 4, 3],
       [2, 1, 5]])

```

Examples

```

-----
>>> a = np.array([[1,2,3], [4,5,6]])
>>> np.reshape(a, 6)
array([1, 2, 3, 4, 5, 6])
>>> np.reshape(a, 6, order='F')
array([1, 4, 2, 5, 3, 6])

>>> np.reshape(a, (3,-1))      # the unspecified value is inferred to be 2
array([[1, 2],
       [3, 4],
       [5, 6]])
"""
return _wrapfunc(a, 'reshape', newshape, order=order)
```

```

def _choose_dispatcher(a, choices, out=None, mode=None):
    yield a
    yield from choices
    yield out

```

```

@array_function_dispatch(_choose_dispatcher)
def choose(a, choices, out=None, mode='raise'):
    """
    Construct an array from an index array and a set of arrays to choose from.

    First of all, if confused or uncertain, definitely look at the Examples -
    in its full generality, this function is less simple than it might
    seem from the following code description (below ndi =
    `numpy.lib.index_tricks`):

    ``np.choose(a,c) == np.array([c[a[I]][I] for I in ndi.ndindex(a.shape)])``.

```

But this omits some subtleties. Here is a fully general summary:

Given an "index" array (`a`) of integers and a sequence of `n` arrays (`choices`), `a` and each choice array are first broadcast, as necessary, to arrays of a common shape; calling these *Ba* and *Bchoices[i], i = 0,...,n-1* we have that, necessarily, ``Ba.shape == Bchoices[i].shape`` for each `i`. Then, a new array with shape ``Ba.shape`` is created as follows:

- * if ``mode=raise`` (the default), then, first of all, each element of `a` (and thus `Ba`) must be in the range `[0, n-1]`; now, suppose that `i` (in that range) is the value at the `(j0, j1, ..., jm)` position in `Ba` - then the value at the same position in the new array is the value in `Bchoices[i]` at that same position;
- * if ``mode=wrap``, values in `a` (and thus `Ba`) may be any (signed) integer; modular arithmetic is used to map integers outside the range `[0, n-1]` back into that range; and then the new array is constructed as above;
- * if ``mode=clip``, values in `a` (and thus `Ba`) may be any (signed) integer; negative integers are mapped to 0; values greater than `n-1` are mapped to `n-1`; and then the new array is constructed as above.

Parameters

`a : int array`

This array must contain integers in `[0, n-1]`, where `n` is the number of choices, unless ``mode=wrap`` or ``mode=clip``, in which cases any integers are permissible.

`choices : sequence of arrays`

Choice arrays. `a` and all of the choices must be broadcastable to the same shape. If `choices` is itself an array (not recommended), then its outermost dimension (i.e., the one corresponding to ``choices.shape[0]``) is taken as defining the "sequence".

`out : array, optional`

If provided, the result will be inserted into this array. It should be of the appropriate shape and dtype. Note that `out` is always buffered if `mode='raise'`; use other modes for better performance.

`mode : {'raise' (default), 'wrap', 'clip'}, optional`

Specifies how indices outside `[0, n-1]` will be treated:

- * 'raise' : an exception is raised
- * 'wrap' : value becomes value mod `n`
- * 'clip' : values < 0 are mapped to 0, values > n-1 are mapped to n-1

Returns

`merged_array : array`
The merged result.

```

Raises
-----
ValueError: shape mismatch
    If `a` and each choice array are not all broadcastable to the same
    shape.

See Also
-----
ndarray.choose : equivalent method
numpy.take_along_axis : Preferable if `choices` is an array

Notes
-----
To reduce the chance of misinterpretation, even though the following
"abuse" is nominally supported, `choices` should neither be, nor be
thought of as, a single array, i.e., the outermost sequence-like container
should be either a list or a tuple.

Examples
-----
>>> choices = [[0, 1, 2, 3], [10, 11, 12, 13],
...     [20, 21, 22, 23], [30, 31, 32, 33]]
>>> np.choose([2, 3, 1, 0], choices
... # the first element of the result will be the first element of the
... # third (2+1) "array" in choices, namely, 20; the second element
... # will be the second element of the fourth (3+1) choice array, i.e.,
... # 31, etc.
... )
array([20, 31, 12, 3])
>>> np.choose([2, 4, 1, 0], choices, mode='clip') # 4 goes to 3 (4-1)
array([20, 31, 12, 3])
>>> # because there are 4 choice arrays
>>> np.choose([2, 4, 1, 0], choices, mode='wrap') # 4 goes to (4 mod 4)
array([20, 1, 12, 3])
>>> # i.e., 0

A couple examples illustrating how choose broadcasts:

>>> a = [[1, 0, 1], [0, 1, 0], [1, 0, 1]]
>>> choices = [-10, 10]
>>> np.choose(a, choices)
array([[ 10, -10,  10],
       [-10,  10, -10],
       [ 10, -10,  10]])

>>> # With thanks to Anne Archibald
>>> a = np.array([0, 1]).reshape((2,1,1))
>>> c1 = np.array([1, 2, 3]).reshape((1,3,1))
>>> c2 = np.array([-1, -2, -3, -4, -5]).reshape((1,1,5))
>>> np.choose(a, (c1, c2)) # result is 2x3x5, res[0,:,:]=c1, res[1,:,:]=c2
array([[[ 1,  1,  1,  1,  1],
        [ 2,  2,  2,  2,  2],
        [ 3,  3,  3,  3,  3]],
       [[-1, -2, -3, -4, -5],
        [-1, -2, -3, -4, -5],
        [-1, -2, -3, -4, -5]]])

"""
return _wrapfunc(a, 'choose', choices, out=out, mode=mode)

def _repeat_dispatcher(a, repeats, axis=None):

```

```
return (a,)

@array_function_dispatch(_repeat_dispatcher)
def repeat(a, repeats, axis=None):
    """
    Repeat elements of an array.

    Parameters
    -----
    a : array_like
        Input array.
    repeats : int or array of ints
        The number of repetitions for each element. `repeats` is broadcasted
        to fit the shape of the given axis.
    axis : int, optional
        The axis along which to repeat values. By default, use the
        flattened input array, and return a flat output array.

    Returns
    -----
    repeated_array : ndarray
        Output array which has the same shape as `a`, except along
        the given axis.

    See Also
    -----
    tile : Tile an array.

    Examples
    -----
    >>> np.repeat(3, 4)
    array([3, 3, 3, 3])
    >>> x = np.array([[1,2],[3,4]])
    >>> np.repeat(x, 2)
    array([1, 1, 2, 2, 3, 3, 4, 4])
    >>> np.repeat(x, 3, axis=1)
    array([[1, 1, 1, 2, 2, 2],
           [3, 3, 3, 4, 4, 4]])
    >>> np.repeat(x, [1, 2], axis=0)
    array([[1, 2],
           [3, 4],
           [3, 4]])

    """
    return _wrapfunc(a, 'repeat', repeats, axis=axis)

def _put_dispatcher(a, ind, v, mode=None):
    return (a, ind, v)

@array_function_dispatch(_put_dispatcher)
def put(a, ind, v, mode='raise'):
    """
    Replaces specified elements of an array with given values.

    The indexing works on the flattened target array. `put` is roughly
    equivalent to:

    ::

        a.flat[ind] = v
```

```
Parameters
-----
a : ndarray
    Target array.
ind : array_like
    Target indices, interpreted as integers.
v : array_like
    Values to place in `a` at target indices. If `v` is shorter than
    `ind` it will be repeated as necessary.
mode : {'raise', 'wrap', 'clip'}, optional
    Specifies how out-of-bounds indices will behave.

    * 'raise' -- raise an error (default)
    * 'wrap' -- wrap around
    * 'clip' -- clip to the range

'clip' mode means that all indices that are too large are replaced
by the index that addresses the last element along that axis. Note
that this disables indexing with negative numbers. In 'raise' mode,
if an exception occurs the target array may still be modified.
```

See Also

```
-----
putmask, place
put_along_axis : Put elements by matching the array and the index arrays
```

Examples

```
-----
>>> a = np.arange(5)
>>> np.put(a, [0, 2], [-44, -55])
>>> a
array([-44,     1, -55,     3,     4])

>>> a = np.arange(5)
>>> np.put(a, 22, -5, mode='clip')
>>> a
array([ 0,   1,   2,   3, -5])
```

"""

```
try:
    put = a.put
except AttributeError:
    raise TypeError("argument 1 must be numpy.ndarray, "
                    "not {}".format(name=type(a).__name__))
return put(ind, v, mode=mode)
```

```
def _swapaxes_dispatcher(a, axis1, axis2):
    return (a,
```

```
@array_function_dispatch(_swapaxes_dispatcher)
def swapaxes(a, axis1, axis2):
    """
    Interchange two axes of an array.
```

Parameters

```
-----
a : array_like
    Input array.
axis1 : int
```

```

    First axis.
axis2 : int
    Second axis.

Returns
-----
a_swapped : ndarray
    For NumPy >= 1.10.0, if `a` is an ndarray, then a view of `a` is
    returned; otherwise a new array is created. For earlier NumPy
    versions a view of `a` is returned only if the order of the
    axes is changed, otherwise the input array is returned.

Examples
-----
>>> x = np.array([[1,2,3]])
>>> np.swapaxes(x,0,1)
array([[1],
       [2],
       [3]])

>>> x = np.array([[[0,1],[2,3]],[[4,5],[6,7]]])
>>> x
array([[[0, 1],
       [2, 3]],
      [[4, 5],
       [6, 7]]])

>>> np.swapaxes(x,0,2)
array([[[0, 4],
       [2, 6]],
      [[1, 5],
       [3, 7]]])

"""
return _wrapfunc(a, 'swapaxes', axis1, axis2)

def _transpose_dispatcher(a, axes=None):
    return (a,)

@array_function_dispatch(_transpose_dispatcher)
def transpose(a, axes=None):
    """
    Reverse or permute the axes of an array; returns the modified array.

    For an array a with two axes, transpose(a) gives the matrix transpose.

Parameters
-----
a : array_like
    Input array.
axes : tuple or list of ints, optional
    If specified, it must be a tuple or list which contains a permutation of
    [0,1,...,N-1] where N is the number of axes of a. The i'th axis of the
    returned array will correspond to the axis numbered ``axes[i]`` of the
    input. If not specified, defaults to ``range(a.ndim)[:-1]``, which
    reverses the order of the axes.

Returns
-----
p : ndarray
    `a` with its axes permuted. A view is returned whenever

```

possible.

See Also

`moveaxis`

`argsort`

Notes

Use ``transpose(a, argsort(axes))`` to invert the transposition of tensors when using the ``axes`` keyword argument.

Transposing a 1-D array returns an unchanged view of the original array.

Examples

```
>>> x = np.arange(4).reshape((2,2))
>>> x
array([[0, 1],
       [2, 3]])

>>> np.transpose(x)
array([[0, 2],
       [1, 3]])

>>> x = np.ones((1, 2, 3))
>>> np.transpose(x, (1, 0, 2)).shape
(2, 1, 3)

"""
return _wrapfunc(a, 'transpose', axes)
```

```
def _partition_dispatcher(a, kth, axis=None, kind=None, order=None):
    return (a,)
```

```
@array_function_dispatch(_partition_dispatcher)
def partition(a, kth, axis=-1, kind='introselect', order=None):
    """
    Return a partitioned copy of an array.
```

Creates a copy of the array with its elements rearranged in such a way that the value of the element in k-th position is in the position it would be in a sorted array. All elements smaller than the k-th element are moved before this element and all equal or greater are moved behind it. The ordering of the elements in the two partitions is undefined.

.. versionadded:: 1.8.0

Parameters

`a : array_like`

Array to be sorted.

`kth : int or sequence of ints`

Element index to partition by. The k-th value of the element will be in its final sorted position and all smaller elements will be moved before it and all equal or greater elements behind it. The order of all elements in the partitions is undefined. If provided with a sequence of k-th it will partition all elements indexed by k-th of them into their sorted position at once.

`axis : int or None, optional`

```
    Axis along which to sort. If None, the array is flattened before
    sorting. The default is -1, which sorts along the last axis.
kind : {'introselect'}, optional
    Selection algorithm. Default is 'introselect'.
order : str or list of str, optional
    When `a` is an array with fields defined, this argument
    specifies which fields to compare first, second, etc. A single
    field can be specified as a string. Not all fields need be
    specified, but unspecified fields will still be used, in the
    order in which they come up in the dtype, to break ties.
```

Returns

```
partitioned_array : ndarray
    Array of the same type and shape as `a`.
```

See Also

```
ndarray.partition : Method to sort an array in-place.
argpartition : Indirect partition.
sort : Full sorting
```

Notes

The various selection algorithms are characterized by their average speed, worst case performance, work space size, and whether they are stable. A stable sort keeps items with the same key in the same relative order. The available algorithms have the following properties:

| kind | speed | worst case | work space | stable |
|---------------|-------|------------|------------|--------|
| 'introselect' | 1 | $O(n)$ | 0 | no |

All the partition algorithms make temporary copies of the data when partitioning along any but the last axis. Consequently, partitioning along the last axis is faster and uses less space than partitioning along any other axis.

The sort order for complex numbers is lexicographic. If both the real and imaginary parts are non-nan then the order is determined by the real parts except when they are equal, in which case the order is determined by the imaginary parts.

Examples

```
>>> a = np.array([3, 4, 2, 1])
>>> np.partition(a, 3)
array([2, 1, 3, 4])

>>> np.partition(a, (1, 3))
array([1, 2, 3, 4])

"""
if axis is None:
    # flatten returns (1, N) for np.matrix, so always use the last axis
    a = asanyarray(a).flatten()
    axis = -1
else:
    a = asanyarray(a).copy(order="K")
a.partition(kth, axis=axis, kind=kind, order=order)
```

```
    return a

def _argpartition_dispatcher(a, kth, axis=None, kind=None, order=None):
    return (a,)

array_function_dispatch(_argpartition_dispatcher)
def argpartition(a, kth, axis=-1, kind='introselect', order=None):
    """
    Perform an indirect partition along the given axis using the
    algorithm specified by the `kind` keyword. It returns an array of
    indices of the same shape as `a` that index data along the given
    axis in partitioned order.

    .. versionadded:: 1.8.0

    Parameters
    -----
    a : array_like
        Array to sort.
    kth : int or sequence of ints
        Element index to partition by. The k-th element will be in its
        final sorted position and all smaller elements will be moved
        before it and all larger elements behind it. The order all
        elements in the partitions is undefined. If provided with a
        sequence of k-th it will partition all of them into their sorted
        position at once.
    axis : int or None, optional
        Axis along which to sort. The default is -1 (the last axis). If
        None, the flattened array is used.
    kind : {'introselect'}, optional
        Selection algorithm. Default is 'introselect'
    order : str or list of str, optional
        When `a` is an array with fields defined, this argument
        specifies which fields to compare first, second, etc. A single
        field can be specified as a string, and not all fields need be
        specified, but unspecified fields will still be used, in the
        order in which they come up in the dtype, to break ties.

    Returns
    -----
    index_array : ndarray, int
        Array of indices that partition `a` along the specified axis.
        If `a` is one-dimensional, ``a[index_array]`` yields a partitioned `a`.
        More generally, ``np.take_along_axis(a, index_array, axis=a)`` always
        yields the partitioned `a`, irrespective of dimensionality.

    See Also
    -----
    partition : Describes partition algorithms used.
    ndarray.partition : Inplace partition.
    argsort : Full indirect sort.
    take_along_axis : Apply ``index_array`` from argpartition
                      to an array as if by calling partition.

    Notes
    -----
    See `partition` for notes on the different selection algorithms.

    Examples
    -----
    One dimensional array:
```

```
>>> x = np.array([3, 4, 2, 1])
>>> x[np.argpartition(x, 3)]
array([2, 1, 3, 4])
>>> x[np.argpartition(x, (1, 3))]
array([1, 2, 3, 4])

>>> x = [3, 4, 2, 1]
>>> np.array(x)[np.argpartition(x, 3)]
array([2, 1, 3, 4])

Multi-dimensional array:

>>> x = np.array([[3, 4, 2], [1, 3, 1]])
>>> index_array = np.argpartition(x, kth=1, axis=-1)
>>> np.take_along_axis(x, index_array, axis=-1) # same as np.partition(x, kth=1)
array([[2, 3, 4],
       [1, 1, 3]])

"""
return _wrapfunc(a, 'argpartition', kth, axis=axis, kind=kind, order=order)

def _sort_dispatcher(a, axis=None, kind=None, order=None):
    return (a,)

@array_function_dispatch(_sort_dispatcher)
def sort(a, axis=-1, kind=None, order=None):
    """
    Return a sorted copy of an array.

    Parameters
    -----
    a : array_like
        Array to be sorted.
    axis : int or None, optional
        Axis along which to sort. If None, the array is flattened before
        sorting. The default is -1, which sorts along the last axis.
    kind : {'quicksort', 'mergesort', 'heapsort', 'stable'}, optional
        Sorting algorithm. The default is 'quicksort'. Note that both 'stable'
        and 'mergesort' use timsort or radix sort under the covers and, in general,
        the actual implementation will vary with data type. The 'mergesort' option
        is retained for backwards compatibility.

    .. versionchanged:: 1.15.0.
        The 'stable' option was added.

    order : str or list of str, optional
        When `a` is an array with fields defined, this argument specifies
        which fields to compare first, second, etc. A single field can
        be specified as a string, and not all fields need be specified,
        but unspecified fields will still be used, in the order in which
        they come up in the dtype, to break ties.

    Returns
    -----
    sorted_array : ndarray
        Array of the same type and shape as `a`.

    See Also
    -----
    ndarray.sort : Method to sort an array in-place.
```

```
argsort : Indirect sort.  
lexsort : Indirect stable sort on multiple keys.  
searchsorted : Find elements in a sorted array.  
partition : Partial sort.
```

Notes

The various sorting algorithms are characterized by their average speed, worst case performance, work space size, and whether they are stable. A stable sort keeps items with the same key in the same relative order. The four algorithms implemented in NumPy have the following properties:

| kind | speed | worst case | work space | stable |
|-------------|-------|----------------|------------|--------|
| 'quicksort' | 1 | $O(n^2)$ | 0 | no |
| 'heapsort' | 3 | $O(n \log(n))$ | 0 | no |
| 'mergesort' | 2 | $O(n \log(n))$ | $\sim n/2$ | yes |
| 'timsort' | 2 | $O(n \log(n))$ | $\sim n/2$ | yes |

.. note:: The datatype determines which of 'mergesort' or 'timsort' is actually used, even if 'mergesort' is specified. User selection at a finer scale is not currently available.

All the sort algorithms make temporary copies of the data when sorting along any but the last axis. Consequently, sorting along the last axis is faster and uses less space than sorting along any other axis.

The sort order for complex numbers is lexicographic. If both the real and imaginary parts are non-nan then the order is determined by the real parts except when they are equal, in which case the order is determined by the imaginary parts.

Previous to numpy 1.4.0 sorting real and complex arrays containing nan values led to undefined behaviour. In numpy versions $\geq 1.4.0$ nan values are sorted to the end. The extended sort order is:

```
* Real: [R, nan]  
* Complex: [R + Rj, R + nanj, nan + Rj, nan + nanj]
```

where R is a non-nan real value. Complex values with the same nan placements are sorted according to the non-nan part if it exists. Non-nan values are sorted as before.

.. versionadded:: 1.12.0

```
quicksort has been changed to `introsort  
<https://en.wikipedia.org/wiki/Introsort>`_.  
When sorting does not make enough progress it switches to  
'heapsort <https://en.wikipedia.org/wiki/Heapsort>`_.  
This implementation makes quicksort  $O(n \log(n))$  in the worst case.
```

'stable' automatically chooses the best stable sorting algorithm for the data type being sorted.

It, along with 'mergesort' is currently mapped to
'timsort <<https://en.wikipedia.org/wiki/Timsort>>`_
or 'radix sort <https://en.wikipedia.org/wiki/Radix_sort>`_
depending on the data type.

API forward compatibility currently limits the ability to select the implementation and it is hardwired for the different

```

data types.

.. versionadded:: 1.17.0

Timsort is added for better performance on already or nearly
sorted data. On random data timsort is almost identical to
mergesort. It is now used for stable sort while quicksort is still the
default sort if none is chosen. For timsort details, refer to
`CPython listsort.txt
<https://github.com/python/cpython/blob/3.7/Objects/listsort.txt>`.
'mergesort' and 'stable' are mapped to radix sort for integer data types. Radix sort
is an
 $O(n)$  sort instead of  $O(n \log n)$ .

.. versionchanged:: 1.18.0

NaT now sorts to the end of arrays for consistency with NaN.

Examples
-----
>>> a = np.array([[1,4],[3,1]])
>>> np.sort(a)                      # sort along the last axis
array([[1, 4],
       [1, 3]])
>>> np.sort(a, axis=None)          # sort the flattened array
array([1, 1, 3, 4])
>>> np.sort(a, axis=0)             # sort along the first axis
array([[1, 1],
       [3, 4]])

Use the `order` keyword to specify a field to use when sorting a
structured array:

>>> dtype = [('name', 'S10'), ('height', float), ('age', int)]
>>> values = [('Arthur', 1.8, 41), ('Lancelot', 1.9, 38),
...             ('Galahad', 1.7, 38)]
>>> a = np.array(values, dtype=dtype)      # create a structured array
>>> np.sort(a, order='height')            # doctest: +SKIP
array([('Galahad', 1.7, 38), ('Arthur', 1.8, 41),
       ('Lancelot', 1.8999999999999999, 38)],
      dtype=[('name', '|S10'), ('height', '<f8'), ('age', '<i4')])

Sort by age, then height if ages are equal:

>>> np.sort(a, order=['age', 'height'])      # doctest: +SKIP
array([('Galahad', 1.7, 38), ('Lancelot', 1.8999999999999999, 38),
       ('Arthur', 1.8, 41)],
      dtype=[('name', '|S10'), ('height', '<f8'), ('age', '<i4')])

"""
if axis is None:
    # flatten returns (1, N) for np.matrix, so always use the last axis
    a = asanyarray(a).flatten()
    axis = -1
else:
    a = asanyarray(a).copy(order="K")
    a.sort(axis=axis, kind=kind, order=order)
return a

def _argsort_dispatcher(a, axis=None, kind=None, order=None):
    return (a,)
```

```
@array_function_dispatch(_argsort_dispatcher)
def argsort(a, axis=-1, kind=None, order=None):
    """
    Returns the indices that would sort an array.

    Perform an indirect sort along the given axis using the algorithm specified
    by the `kind` keyword. It returns an array of indices of the same shape as
    `a` that index data along the given axis in sorted order.

    Parameters
    -----
    a : array_like
        Array to sort.
    axis : int or None, optional
        Axis along which to sort. The default is -1 (the last axis). If None,
        the flattened array is used.
    kind : {'quicksort', 'mergesort', 'heapsort', 'stable'}, optional
        Sorting algorithm. The default is 'quicksort'. Note that both 'stable'
        and 'mergesort' use timsort under the covers and, in general, the
        actual implementation will vary with data type. The 'mergesort' option
        is retained for backwards compatibility.

    .. versionchanged:: 1.15.0.
        The 'stable' option was added.
    order : str or list of str, optional
        When `a` is an array with fields defined, this argument specifies
        which fields to compare first, second, etc. A single field can
        be specified as a string, and not all fields need be specified,
        but unspecified fields will still be used, in the order in which
        they come up in the dtype, to break ties.

    Returns
    -----
    index_array : ndarray, int
        Array of indices that sort `a` along the specified `axis`.
        If `a` is one-dimensional, ``a[index_array]`` yields a sorted `a`.
        More generally, ``np.take_along_axis(a, index_array, axis=axis)``
        always yields the sorted `a`, irrespective of dimensionality.

    See Also
    -----
    sort : Describes sorting algorithms used.
    lexsort : Indirect stable sort with multiple keys.
    ndarray.sort : Inplace sort.
    argpartition : Indirect partial sort.
    take_along_axis : Apply ``index_array`` from argsort
                      to an array as if by calling sort.

    Notes
    -----
    See `sort` for notes on the different sorting algorithms.

    As of NumPy 1.4.0 `argsort` works with real/complex arrays containing
    nan values. The enhanced sort order is documented in `sort`.

    Examples
    -----
    One dimensional array:

    >>> x = np.array([3, 1, 2])
    >>> np.argsort(x)
    array([1, 2, 0])
```

```
Two-dimensional array:
```

```
>>> x = np.array([[0, 3], [2, 2]])
>>> x
array([[0, 3],
       [2, 2]])

>>> ind = np.argsort(x, axis=0) # sorts along first axis (down)
>>> ind
array([[0, 1],
       [1, 0]])
>>> np.take_along_axis(x, ind, axis=0) # same as np.sort(x, axis=0)
array([[0, 2],
       [2, 3]])

>>> ind = np.argsort(x, axis=1) # sorts along last axis (across)
>>> ind
array([[0, 1],
       [0, 1]])
>>> np.take_along_axis(x, ind, axis=1) # same as np.sort(x, axis=1)
array([[0, 3],
       [2, 2]])
```

```
Indices of the sorted elements of a N-dimensional array:
```

```
>>> ind = np.unravel_index(np.argsort(x, axis=None), x.shape)
>>> ind
(array([0, 1, 1, 0]), array([0, 0, 1, 1]))
>>> x[ind] # same as np.sort(x, axis=None)
array([0, 2, 2, 3])
```

```
Sorting with keys:
```

```
>>> x = np.array([(1, 0), (0, 1)], dtype=[('x', '<i4'), ('y', '<i4')])
>>> x
array([(1, 0), (0, 1)],
      dtype=[('x', '<i4'), ('y', '<i4')])

>>> np.argsort(x, order=('x', 'y'))
array([1, 0])

>>> np.argsort(x, order=('y', 'x'))
array([0, 1])

"""
return _wrapfunc(a, 'argsort', axis=axis, kind=kind, order=order)
```

```
def _argmax_dispatcher(a, axis=None, out=None):
    return (a, out)
```

```
@array_function_dispatch(_argmax_dispatcher)
def argmax(a, axis=None, out=None):
    """
    Returns the indices of the maximum values along an axis.
```

```
Parameters
```

```
-----
```

```
a : array_like
    Input array.
axis : int, optional
```

```
    By default, the index is into the flattened array, otherwise
    along the specified axis.
out : array, optional
    If provided, the result will be inserted into this array. It should
    be of the appropriate shape and dtype.

Returns
-----
index_array : ndarray of ints
    Array of indices into the array. It has the same shape as `a.shape`
    with the dimension along `axis` removed.

See Also
-----
ndarray.argmax, argmin
amax : The maximum value along a given axis.
unravel_index : Convert a flat index into an index tuple.
take_along_axis : Apply ``np.expand_dims(index_array, axis)``
                  from argmax to an array as if by calling max.

Notes
-----
In case of multiple occurrences of the maximum values, the indices
corresponding to the first occurrence are returned.

Examples
-----
>>> a = np.arange(6).reshape(2,3) + 10
>>> a
array([[10, 11, 12],
       [13, 14, 15]])
>>> np.argmax(a)
5
>>> np.argmax(a, axis=0)
array([1, 1, 1])
>>> np.argmax(a, axis=1)
array([2, 2])

Indexes of the maximal elements of a N-dimensional array:

>>> ind = np.unravel_index(np.argmax(a, axis=None), a.shape)
>>> ind
(1, 2)
>>> a[ind]
15

>>> b = np.arange(6)
>>> b[1] = 5
>>> b
array([0, 5, 2, 3, 4, 5])
>>> np.argmax(b) # Only the first occurrence is returned.
1

>>> x = np.array([[4,2,3], [1,0,3]])
>>> index_array = np.argmax(x, axis=-1)
>>> # Same as np.max(x, axis=-1, keepdims=True)
>>> np.take_along_axis(x, np.expand_dims(index_array, axis=-1), axis=-1)
array([[4],
       [3]])
>>> # Same as np.max(x, axis=-1)
>>> np.take_along_axis(x, np.expand_dims(index_array, axis=-1), axis=-1).squeeze(axis=-1)
array([4, 3])
```

```
"""
    return _wrapfunc(a, 'argmax', axis=axis, out=out)

def _argmin_dispatcher(a, axis=None, out=None):
    return (a, out)

@array_function_dispatch(_argmin_dispatcher)
def argmin(a, axis=None, out=None):
    """
    Returns the indices of the minimum values along an axis.

    Parameters
    ----------
    a : array_like
        Input array.
    axis : int, optional
        By default, the index is into the flattened array, otherwise
        along the specified axis.
    out : array, optional
        If provided, the result will be inserted into this array. It should
        be of the appropriate shape and dtype.

    Returns
    -------
    index_array : ndarray of ints
        Array of indices into the array. It has the same shape as `a.shape`
        with the dimension along `axis` removed.

    See Also
    --------
    ndarray.argmin, argmax
    amin : The minimum value along a given axis.
    unravel_index : Convert a flat index into an index tuple.
    take_along_axis : Apply ``np.expand_dims(index_array, axis)``
        from argmin to an array as if by calling min.

    Notes
    -----
    In case of multiple occurrences of the minimum values, the indices
    corresponding to the first occurrence are returned.

    Examples
    --------
    >>> a = np.arange(6).reshape(2,3) + 10
    >>> a
    array([[10, 11, 12],
           [13, 14, 15]])
    >>> np.argmin(a)
    0
    >>> np.argmin(a, axis=0)
    array([0, 0, 0])
    >>> np.argmin(a, axis=1)
    array([0, 0])

    Indices of the minimum elements of a N-dimensional array:

    >>> ind = np.unravel_index(np.argmin(a, axis=None), a.shape)
    >>> ind
    (0, 0)
    >>> a[ind]
```

```

10

>>> b = np.arange(6) + 10
>>> b[4] = 10
>>> b
array([10, 11, 12, 13, 10, 15])
>>> np.argmin(b) # Only the first occurrence is returned.
0

>>> x = np.array([[4,2,3], [1,0,3]])
>>> index_array = np.argmax(x, axis=-1)
>>> # Same as np.min(x, axis=-1, keepdims=True)
>>> np.take_along_axis(x, np.expand_dims(index_array, axis=-1), axis=-1)
array([[2],
       [0]])
>>> # Same as np.max(x, axis=-1)
>>> np.take_along_axis(x, np.expand_dims(index_array, axis=-1), axis=-1).squeeze(axis=-1)
array([2, 0])

"""
return _wrapfunc(a, 'argmin', axis=axis, out=out)

def _searchsorted_dispatcher(a, v, side=None, sorter=None):
    return (a, v, sorter)

@array_function_dispatch(_searchsorted_dispatcher)
def searchsorted(a, v, side='left', sorter=None):
    """
    Find indices where elements should be inserted to maintain order.

    Find the indices into a sorted array `a` such that, if the
    corresponding elements in `v` were inserted before the indices, the
    order of `a` would be preserved.

    Assuming that `a` is sorted:

    =====
    `side`    returned index `i` satisfies
    =====
    left     ``a[i-1] < v <= a[i]``
    right    ``a[i-1] <= v < a[i]``
    =====

    Parameters
    -----
    a : 1-D array_like
        Input array. If `sorter` is None, then it must be sorted in
        ascending order, otherwise `sorter` must be an array of indices
        that sort it.
    v : array_like
        Values to insert into `a`.
    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 `a`).
    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.

    .. versionadded:: 1.7.0

```

```
Returns
-----
indices : array of ints
    Array of insertion points with the same shape as `v`.

See Also
-----
sort : Return a sorted copy of an array.
histogram : Produce histogram from 1-D data.

Notes
-----
Binary search is used to find the required insertion points.

As of NumPy 1.4.0 `searchsorted` works with real/complex arrays containing
`nan` values. The enhanced sort order is documented in `sort`.

This function uses the same algorithm as the builtin python `bisect.bisect_left` (
``side='left'``) and `bisect.bisect_right` (``side='right'``) functions,
which is also vectorized in the `v` argument.

Examples
-----
>>> np.searchsorted([1,2,3,4,5], 3)
2
>>> np.searchsorted([1,2,3,4,5], 3, side='right')
3
>>> np.searchsorted([1,2,3,4,5], [-10, 10, 2, 3])
array([0, 5, 1, 2])

"""
return _wrapfunc(a, 'searchsorted', v, side=side, sorter=sorter)

def _resize_dispatcher(a, new_shape):
    return (a,)

@array_function_dispatch(_resize_dispatcher)
def resize(a, new_shape):
    """
    Return a new array with the specified shape.

    If the new array is larger than the original array, then the new
    array is filled with repeated copies of `a`. Note that this behavior
    is different from a.resize(new_shape) which fills with zeros instead
    of repeated copies of `a`.

    Parameters
    -----
    a : array_like
        Array to be resized.

    new_shape : int or tuple of int
        Shape of resized array.

    Returns
    -----
    reshaped_array : ndarray
        The new array is formed from the data in the old array, repeated
        if necessary to fill out the required number of elements. The
        data are repeated in the order that they are stored in memory.
```

See Also

ndarray.resize : resize an array in-place.

Notes

Warning: This functionality does ****not**** consider axes separately, i.e. it does not apply interpolation/extrapolation.

It fills the return array with the required number of elements, taken from `a` as they are laid out in memory, disregarding strides and axes. (This is in case the new shape is smaller. For larger, see above.) This functionality is therefore not suitable to resize images, or data where each axis represents a separate and distinct entity.

Examples

```
>>> a=np.array([[0,1],[2,3]])
>>> np.resize(a,(2,3))
array([[0, 1, 2],
       [3, 0, 1]])
>>> np.resize(a,(1,4))
array([[0, 1, 2, 3]])
>>> np.resize(a,(2,4))
array([[0, 1, 2, 3],
       [0, 1, 2, 3]])
```

"""

```
if isinstance(new_shape, (int, nt.integer)):
    new_shape = (new_shape,)
a = ravel(a)
Na = len(a)
total_size = um.multiply.reduce(new_shape)
if Na == 0 or total_size == 0:
    return mu.zeros(new_shape, a.dtype)

n_copies = int(total_size / Na)
extra = total_size % Na

if extra != 0:
    n_copies = n_copies + 1
    extra = Na - extra

a = concatenate((a,) * n_copies)
if extra > 0:
    a = a[:-extra]

return reshape(a, new_shape)
```

```
def _squeeze_dispatcher(a, axis=None):
    return (a,)
```

```
@array_function_dispatch(_squeeze_dispatcher)
def squeeze(a, axis=None):
    """
```

Remove single-dimensional entries from the shape of an array.

Parameters

a : array_like
Input data.

```
axis : None or int or tuple of ints, optional
.. versionadded:: 1.7.0

    Selects a subset of the single-dimensional entries in the
    shape. If an axis is selected with shape entry greater than
    one, an error is raised.

Returns
-----
squeezed : ndarray
    The input array, but with all or a subset of the
    dimensions of length 1 removed. This is always `a` itself
    or a view into `a`. Note that if all axes are squeezed,
    the result is a 0d array and not a scalar.

Raises
-----
ValueError
    If `axis` is not None, and an axis being squeezed is not of length 1

See Also
-----
expand_dims : The inverse operation, adding singleton dimensions
reshape : Insert, remove, and combine dimensions, and resize existing ones

Examples
-----
>>> x = np.array([[[0], [1], [2]]])
>>> x.shape
(1, 3, 1)
>>> np.squeeze(x).shape
(3,)
>>> np.squeeze(x, axis=0).shape
(3, 1)
>>> np.squeeze(x, axis=1).shape
Traceback (most recent call last):
...
ValueError: cannot select an axis to squeeze out which has size not equal to one
>>> np.squeeze(x, axis=2).shape
(1, 3)
>>> x = np.array([[1234]])
>>> x.shape
(1, 1)
>>> np.squeeze(x)
array(1234) # 0d array
>>> np.squeeze(x).shape
()
>>> np.squeeze(x)[()]
1234

"""
try:
    squeeze = a.squeeze
except AttributeError:
    return _wrapit(a, 'squeeze', axis=axis)
if axis is None:
    return squeeze()
else:
    return squeeze(axis=axis)

def _diagonal_dispatcher(a, offset=None, axis1=None, axis2=None):
    return (a,)
```

```
@array_function_dispatch(_diagonal_dispatcher)
def diagonal(a, offset=0, axis1=0, axis2=1):
    """
    Return specified diagonals.

    If `a` is 2-D, returns the diagonal of `a` with the given offset,
    i.e., the collection of elements of the form ``a[i, i+offset]``. If
    `a` has more than two dimensions, then the axes specified by `axis1`
    and `axis2` are used to determine the 2-D sub-array whose diagonal is
    returned. The shape of the resulting array can be determined by
    removing `axis1` and `axis2` and appending an index to the right equal
    to the size of the resulting diagonals.

    In versions of NumPy prior to 1.7, this function always returned a new,
    independent array containing a copy of the values in the diagonal.

    In NumPy 1.7 and 1.8, it continues to return a copy of the diagonal,
    but depending on this fact is deprecated. Writing to the resulting
    array continues to work as it used to, but a FutureWarning is issued.

    Starting in NumPy 1.9 it returns a read-only view on the original array.
    Attempting to write to the resulting array will produce an error.

    In some future release, it will return a read/write view and writing to
    the returned array will alter your original array. The returned array
    will have the same type as the input array.

    If you don't write to the array returned by this function, then you can
    just ignore all of the above.

    If you depend on the current behavior, then we suggest copying the
    returned array explicitly, i.e., use ``np.diagonal(a).copy()`` instead
    of just ``np.diagonal(a)``. This will work with both past and future
    versions of NumPy.

Parameters
-----
a : array_like
    Array from which the diagonals are taken.
offset : int, optional
    Offset of the diagonal from the main diagonal. Can be positive or
    negative. Defaults to main diagonal (0).
axis1 : int, optional
    Axis to be used as the first axis of the 2-D sub-arrays from which
    the diagonals should be taken. Defaults to first axis (0).
axis2 : int, optional
    Axis to be used as the second axis of the 2-D sub-arrays from
    which the diagonals should be taken. Defaults to second axis (1).

Returns
-----
array_of_diagonals : ndarray
    If `a` is 2-D, then a 1-D array containing the diagonal and of the
    same type as `a` is returned unless `a` is a `matrix`, in which case
    a 1-D array rather than a (2-D) `matrix` is returned in order to
    maintain backward compatibility.

    If ``a.ndim > 2``, then the dimensions specified by `axis1` and `axis2`
    are removed, and a new axis inserted at the end corresponding to the
    diagonal.
```

```
Raises
-----
ValueError
    If the dimension of `a` is less than 2.

See Also
-----
diag : MATLAB work-a-like for 1-D and 2-D arrays.
diagflat : Create diagonal arrays.
trace : Sum along diagonals.
```

```
Examples
-----
>>> a = np.arange(4).reshape(2,2)
>>> a
array([[0, 1],
       [2, 3]])
>>> a.diagonal()
array([0, 3])
>>> a.diagonal(1)
array([1])
```

A 3-D example:

```
>>> a = np.arange(8).reshape(2,2,2); a
array([[[0, 1],
        [2, 3]],
       [[4, 5],
        [6, 7]]])
>>> a.diagonal(0, # Main diagonals of two arrays created by skipping
...             0, # across the outer(left)-most axis last and
...             1) # the "middle" (row) axis first.
array([[0, 6],
       [1, 7]])
```

The sub-arrays whose main diagonals we just obtained; note that each corresponds to fixing the right-most (column) axis, and that the diagonals are "packed" in rows.

```
>>> a[:, :, 0] # main diagonal is [0 6]
array([[0, 2],
       [4, 6]])
>>> a[:, :, 1] # main diagonal is [1 7]
array([[1, 3],
       [5, 7]])
```

The anti-diagonal can be obtained by reversing the order of elements using either `numpy.flipud` or `numpy.fliplr`.

```
>>> a = np.arange(9).reshape(3, 3)
>>> a
array([[0, 1, 2],
       [3, 4, 5],
       [6, 7, 8]])
>>> np.fliplr(a).diagonal() # Horizontal flip
array([2, 4, 6])
>>> np.flipud(a).diagonal() # Vertical flip
array([6, 4, 2])
```

Note that the order in which the diagonal is retrieved varies depending on the flip function.

"""

```
if isinstance(a, np.matrix):
```

```

# Make diagonal of matrix 1-D to preserve backward compatibility.
    return asarray(a).diagonal(offset=offset, axis1=axis1, axis2=axis2)
else:
    return asanyarray(a).diagonal(offset=offset, axis1=axis1, axis2=axis2)

def _trace_dispatcher(
    a, offset=None, axis1=None, axis2=None, dtype=None, out=None):
    return (a, out)

@array_function_dispatch(_trace_dispatcher)
def trace(a, offset=0, axis1=0, axis2=1, dtype=None, out=None):
    """
    Return the sum along diagonals of the array.

    If `a` is 2-D, the sum along its diagonal with the given offset
    is returned, i.e., the sum of elements ``a[i,i+offset]`` for all i.

    If `a` has more than two dimensions, then the axes specified by `axis1` and
    `axis2` are used to determine the 2-D sub-arrays whose traces are returned.
    The shape of the resulting array is the same as that of `a` with `axis1`
    and `axis2` removed.

    Parameters
    -----
    a : array_like
        Input array, from which the diagonals are taken.
    offset : int, optional
        Offset of the diagonal from the main diagonal. Can be both positive
        and negative. Defaults to 0.
    axis1, axis2 : int, optional
        Axes to be used as the first and second axis of the 2-D sub-arrays
        from which the diagonals should be taken. Defaults are the first two
        axes of `a`.
    dtype : dtype, optional
        Determines the data-type of the returned array and of the accumulator
        where the elements are summed. If `dtype` has the value `None` and `a` is
        of integer type of precision less than the default integer
        precision, then the default integer precision is used. Otherwise,
        the precision is the same as that of `a`.
    out : ndarray, optional
        Array into which the output is placed. Its type is preserved and
        it must be of the right shape to hold the output.

    Returns
    -----
    sum_along_diagonals : ndarray
        If `a` is 2-D, the sum along the diagonal is returned. If `a` has
        larger dimensions, then an array of sums along diagonals is returned.

    See Also
    -----
    diag, diagonal, diagflat

    Examples
    -----
    >>> np.trace(np.eye(3))
    3.0
    >>> a = np.arange(8).reshape((2,2,2))
    >>> np.trace(a)
    array([6, 8])

```

```
>>> a = np.arange(24).reshape((2,2,2,3))
>>> np.trace(a).shape
(2, 3)

"""
if isinstance(a, np.matrix):
    # Get trace of matrix via an array to preserve backward compatibility.
    return asarray(a).trace(offset=offset, axis1=axis1, axis2=axis2, dtype=dtype,
out=out)
else:
    return asanyarray(a).trace(offset=offset, axis1=axis1, axis2=axis2, dtype=dtype,
out=out)

def _ravel_dispatcher(a, order=None):
    return (a,)
```

@array_function_dispatch(_ravel_dispatcher)

```
def ravel(a, order='C'):
    """Return a contiguous flattened array.
```

A 1-D array, containing the elements of the input, is returned. A copy is made only if needed.

As of NumPy 1.10, the returned array will have the same type as the input array. (for example, a masked array will be returned for a masked array input)

Parameters

```
a : array_like
    Input array. The elements in `a` are read in the order specified by
    `order`, and packed as a 1-D array.
order : {'C', 'F', 'A', 'K'}, optional
```

The elements of `a` are read using this index order. 'C' means to index the elements in row-major, C-style order, with the last axis index changing fastest, back to the first axis index changing slowest. 'F' means to index the elements in column-major, Fortran-style order, with the first index changing fastest, and the last index changing slowest. Note that the 'C' and 'F' options take no account of the memory layout of the underlying array, and only refer to the order of axis indexing. 'A' means to read the elements in Fortran-like index order if `a` is Fortran *contiguous* in memory, C-like order otherwise. 'K' means to read the elements in the order they occur in memory, except for reversing the data when strides are negative. By default, 'C' index order is used.

Returns

```
y : array_like
    y is an array of the same subtype as `a`, with shape ``a.size,``.
    Note that matrices are special cased for backward compatibility, if `a`
    is a matrix, then y is a 1-D ndarray.
```

See Also

ndarray.flat : 1-D iterator over an array.

ndarray.flatten : 1-D array copy of the elements of an array
 in row-major order.

```
ndarray.reshape : Change the shape of an array without changing its data.
```

Notes

In row-major, C-style order, in two dimensions, the row index varies the slowest, and the column index the quickest. This can be generalized to multiple dimensions, where row-major order implies that the index along the first axis varies slowest, and the index along the last quickest. The opposite holds for column-major, Fortran-style index ordering.

When a view is desired in as many cases as possible, ``arr.reshape(-1)`` may be preferable.

Examples

It is equivalent to ``reshape(-1, order=order)``.

```
>>> x = np.array([[1, 2, 3], [4, 5, 6]])
>>> np.ravel(x)
array([1, 2, 3, 4, 5, 6])
```

```
>>> x.reshape(-1)
array([1, 2, 3, 4, 5, 6])
```

```
>>> np.ravel(x, order='F')
array([1, 4, 2, 5, 3, 6])
```

When ``order`` is 'A', it will preserve the array's 'C' or 'F' ordering:

```
>>> np.ravel(x.T)
array([1, 4, 2, 5, 3, 6])
>>> np.ravel(x.T, order='A')
array([1, 2, 3, 4, 5, 6])
```

When ``order`` is 'K', it will preserve orderings that are neither 'C' nor 'F', but won't reverse axes:

```
>>> a = np.arange(3)[::-1]; a
array([2, 1, 0])
>>> a.ravel(order='C')
array([2, 1, 0])
>>> a.ravel(order='K')
array([2, 1, 0])

>>> a = np.arange(12).reshape(2,3,2).swapaxes(1,2); a
array([[[ 0,  2,  4],
       [ 1,  3,  5]],
      [[ 6,  8, 10],
       [ 7,  9, 11]])]
>>> a.ravel(order='C')
array([ 0,  2,  4,  1,  3,  5,  6,  8, 10,  7,  9, 11])
>>> a.ravel(order='K')
array([ 0,  1,  2,  3,  4,  5,  6,  7,  8,  9, 10, 11])

"""
if isinstance(a, np.matrix):
    return asarray(a).ravel(order=order)
else:
    return asanyarray(a).ravel(order=order)

def _nonzero_dispatcher(a):
```

```
return (a,)

@array_function_dispatch(_nonzero_dispatcher)
def nonzero(a):
    """
    Return the indices of the elements that are non-zero.

    Returns a tuple of arrays, one for each dimension of `a`,
    containing the indices of the non-zero elements in that
    dimension. The values in `a` are always tested and returned in
    row-major, C-style order.

    To group the indices by element, rather than dimension, use `argwhere`,
    which returns a row for each non-zero element.

    .. note::

        When called on a zero-d array or scalar, ``nonzero(a)`` is treated
        as ``nonzero(atleast1d(a))``.

    .. deprecated:: 1.17.0

        Use `atleast1d` explicitly if this behavior is deliberate.

    Parameters
    -----
    a : array_like
        Input array.

    Returns
    -----
    tuple_of_arrays : tuple
        Indices of elements that are non-zero.

    See Also
    -----
    flatnonzero :
        Return indices that are non-zero in the flattened version of the input
        array.
    ndarray.nonzero :
        Equivalent ndarray method.
    count_nonzero :
        Counts the number of non-zero elements in the input array.

    Notes
    -----
    While the nonzero values can be obtained with ``a[nonzero(a)]``, it is
    recommended to use ``x[x.astype(bool)]`` or ``x[x != 0]`` instead, which
    will correctly handle 0-d arrays.

    Examples
    -----
    >>> x = np.array([[3, 0, 0], [0, 4, 0], [5, 6, 0]])
    >>> x
    array([[3, 0, 0],
           [0, 4, 0],
           [5, 6, 0]])
    >>> np.nonzero(x)
    (array([0, 1, 2, 2]), array([0, 1, 0, 1]))

    >>> x[np.nonzero(x)]
    array([3, 4, 5, 6])
```

```
>>> np.transpose(np.nonzero(x))
array([[0, 0],
       [1, 1],
       [2, 0],
       [2, 1]])
```

A common use for ``nonzero`` is to find the indices of an array, where a condition is True. Given an array `a`, the condition `a > 3` is a boolean array and since False is interpreted as 0, `np.nonzero(a > 3)` yields the indices of the `a` where the condition is true.

```
>>> a = np.array([[1, 2, 3], [4, 5, 6], [7, 8, 9]])
>>> a > 3
array([[False, False, False],
       [ True,  True,  True],
       [ True,  True,  True]])
>>> np.nonzero(a > 3)
(array([1, 1, 1, 2, 2, 2]), array([0, 1, 2, 0, 1, 2]))
```

Using this result to index `a` is equivalent to using the mask directly:

```
>>> a[np.nonzero(a > 3)]
array([4, 5, 6, 7, 8, 9])
>>> a[a > 3] # prefer this spelling
array([4, 5, 6, 7, 8, 9])
```

``nonzero`` can also be called as a method of the array.

```
>>> (a > 3).nonzero()
(array([1, 1, 1, 2, 2, 2]), array([0, 1, 2, 0, 1, 2]))
"""
return _wrapfunc(a, 'nonzero')
```

```
def _shape_dispatcher(a):
    return (a,)
```

```
@array_function_dispatch(_shape_dispatcher)
def shape(a):
```

```
    """
    Return the shape of an array.
```

Parameters

a : array_like
Input array.

Returns

shape : tuple of ints

The elements of the shape tuple give the lengths of the corresponding array dimensions.

See Also

alen

ndarray.shape : Equivalent array method.

Examples

```
>>> np.shape(np.eye(3))
```

```
(3, 3)
>>> np.shape([[1, 2]])
(1, 2)
>>> np.shape([0])
(1,)
>>> np.shape(0)
()

>>> a = np.array([(1, 2), (3, 4)], dtype=[('x', 'i4'), ('y', 'i4')])
>>> np.shape(a)
(2,)
>>> a.shape
(2,)

"""
try:
    result = a.shape
except AttributeError:
    result = asarray(a).shape
return result

def _compress_dispatcher(condition, a, axis=None, out=None):
    return (condition, a, out)
```

```
@array_function_dispatch(_compress_dispatcher)
def compress(condition, a, axis=None, out=None):
```

```
"""
Return selected slices of an array along given axis.
```

When working along a given axis, a slice along that axis is returned in `output` for each index where `condition` evaluates to True. When working on a 1-D array, `compress` is equivalent to `extract`.

Parameters

condition : 1-D array of bools

 Array that selects which entries to return. If len(condition) is less than the size of `a` along the given axis, then output is truncated to the length of the condition array.

a : array_like

 Array from which to extract a part.

axis : int, optional

 Axis along which to take slices. If None (default), work on the flattened array.

out : ndarray, optional

 Output array. Its type is preserved and it must be of the right shape to hold the output.

Returns

compressed_array : ndarray

 A copy of `a` without the slices along axis for which `condition` is false.

See Also

[take](#), [choose](#), [diag](#), [diagonal](#), [select](#)

[ndarray.compress](#) : Equivalent method in ndarray

[np.extract](#): Equivalent method when working on 1-D arrays

[ufuncs-output-type](#)

Examples

```
-----
>>> a = np.array([[1, 2], [3, 4], [5, 6]])
>>> a
array([[1, 2],
       [3, 4],
       [5, 6]])
>>> np.compress([0, 1], a, axis=0)
array([[3, 4]])
>>> np.compress([False, True, True], a, axis=0)
array([[3, 4],
       [5, 6]])
>>> np.compress([False, True], a, axis=1)
array([[2],
       [4],
       [6]])
```

Working on the flattened array does not return slices along an axis but selects elements.

```
>>> np.compress([False, True], a)
array([2])

"""
return _wrapfunc(a, 'compress', condition, axis=axis, out=out)
```

```
def _clip_dispatcher(a, a_min, a_max, out=None, **kwargs):
    return (a, a_min, a_max)
```

```
@array_function_dispatch(_clip_dispatcher)
def clip(a, a_min, a_max, out=None, **kwargs):
    """
    Clip (limit) the values in an array.
```

Given an interval, values outside the interval are clipped to the interval edges. For example, if an interval of ``[0, 1]`` is specified, values smaller than 0 become 0, and values larger than 1 become 1.

Equivalent to but faster than ``np.minimum(a_max, np.maximum(a, a_min))``.

No check is performed to ensure ``a_min < a_max``.

Parameters

a : array_like

Array containing elements to clip.

a_min : scalar or array_like or None

Minimum value. If None, clipping is not performed on lower interval edge. Not more than one of `a_min` and `a_max` may be None.

a_max : scalar or array_like or None

Maximum value. If None, clipping is not performed on upper interval edge. Not more than one of `a_min` and `a_max` may be None. If `a_min` or `a_max` are array_like, then the three arrays will be broadcasted to match their shapes.

out : ndarray, optional

The results will be placed in this array. It may be the input array for in-place clipping. `out` must be of the right shape to hold the output. Its type is preserved.

**kwargs

```

For other keyword-only arguments, see the
:ref:`ufunc docs <ufuncs.kwargs>`.

.. versionadded:: 1.17.0

Returns
-----
clipped_array : ndarray
    An array with the elements of `a`, but where values
    < `a_min` are replaced with `a_min`, and those > `a_max`
    with `a_max`.

See Also
-----
ufuncs-output-type

Examples
-----
>>> a = np.arange(10)
>>> np.clip(a, 1, 8)
array([1, 1, 2, 3, 4, 5, 6, 7, 8, 8])
>>> a
array([0, 1, 2, 3, 4, 5, 6, 7, 8, 9])
>>> np.clip(a, 3, 6, out=a)
array([3, 3, 3, 3, 4, 5, 6, 6, 6, 6])
>>> a = np.arange(10)
>>> a
array([0, 1, 2, 3, 4, 5, 6, 7, 8, 9])
>>> np.clip(a, [3, 4, 1, 1, 1, 4, 4, 4, 4, 4], 8)
array([3, 4, 2, 3, 4, 5, 6, 7, 8, 8])

"""
return _wrapfunc(a, 'clip', a_min, a_max, out=out, **kwargs)

def _sum_dispatcher(a, axis=None, dtype=None, out=None, keepdims=None,
                    initial=None, where=None):
    return (a, out)

@array_function_dispatch(_sum_dispatcher)
def sum(a, axis=None, dtype=None, out=None, keepdims=np._NoValue,
        initial=np._NoValue, where=np._NoValue):
    """
    Sum of array elements over a given axis.

Parameters
-----
a : array_like
    Elements to sum.
axis : None or int or tuple of ints, optional
    Axis or axes along which a sum is performed. The default,
    axis=None, will sum all of the elements of the input array. If
    axis is negative it counts from the last to the first axis.

.. versionadded:: 1.7.0

    If axis is a tuple of ints, a sum is performed on all of the axes
    specified in the tuple instead of a single axis or all the axes as
    before.
dtype : dtype, optional
    The type of the returned array and of the accumulator in which the
    elements are summed. The dtype of `a` is used by default unless `a`
```

has an integer dtype of less precision than the default platform integer. In that case, if `a` is signed then the platform integer is used while if `a` is unsigned then an unsigned integer of the same precision as the platform integer is used.

`out` : ndarray, optional
Alternative output array in which to place the result. It must have the same shape as the expected output, but the type of the output values will be cast if necessary.

`keepdims` : bool, optional
If this is set to True, the axes which are reduced are left in the result as dimensions with size one. With this option, the result will broadcast correctly against the input array.

If the default value is passed, then `keepdims` will not be passed through to the `sum` method of sub-classes of `ndarray`, however any non-default value will be. If the sub-class' method does not implement `keepdims` any exceptions will be raised.

`initial` : scalar, optional
Starting value for the sum. See `~numpy.ufunc.reduce` for details.

.. versionadded:: 1.15.0

`where` : array_like of bool, optional
Elements to include in the sum. See `~numpy.ufunc.reduce` for details.

.. versionadded:: 1.17.0

Returns

`sum_along_axis` : ndarray
An array with the same shape as `a`, with the specified axis removed. If `a` is a 0-d array, or if `axis` is None, a scalar is returned. If an output array is specified, a reference to `out` is returned.

See Also

`ndarray.sum` : Equivalent method.
`add.reduce` : Equivalent functionality of `add`.
`cumsum` : Cumulative sum of array elements.
`trapz` : Integration of array values using the composite trapezoidal rule.
`mean, average`

Notes

Arithmetic is modular when using integer types, and no error is raised on overflow.

The sum of an empty array is the neutral element 0:

```
>>> np.sum([])  
0.0
```

For floating point numbers the numerical precision of sum (and `np.add.reduce`) is in general limited by directly adding each number individually to the result causing rounding errors in every step. However, often numpy will use a numerically better approach (partial pairwise summation) leading to improved precision in many use-cases.

This improved precision is always provided when no ``axis`` is given. When ``axis`` is given, it will depend on which axis is summed. Technically, to provide the best speed possible, the improved precision is only used when the summation is along the fast axis in memory. Note that the exact precision may vary depending on other parameters. In contrast to NumPy, Python's ``math.fsum`` function uses a slower but more precise approach to summation. Especially when summing a large number of lower precision floating point numbers, such as ``float32``, numerical errors can become significant. In such cases it can be advisable to use `dtype="float64"` to use a higher precision for the output.

Examples

```
>>> np.sum([0.5, 1.5])
2.0
>>> np.sum([0.5, 0.7, 0.2, 1.5], dtype=np.int32)
1
>>> np.sum([[0, 1], [0, 5]])
6
>>> np.sum([[0, 1], [0, 5]], axis=0)
array([0, 6])
>>> np.sum([[0, 1], [0, 5]], axis=1)
array([1, 5])
>>> np.sum([[0, 1], [np.nan, 5]], where=[False, True], axis=1)
array([1., 5.])
```

If the accumulator is too small, overflow occurs:

```
>>> np.ones(128, dtype=np.int8).sum(dtype=np.int8)
-128
```

You can also start the sum with a value other than zero:

```
>>> np.sum([10], initial=5)
15
"""
if isinstance(a, _gentype):
    # 2018-02-25, 1.15.0
    warnings.warn(
        "Calling np.sum(generator) is deprecated, and in the future will give a
different result."
        "Use np.sum(np.fromiter(generator)) or the python sum builtin instead.",
        DeprecationWarning, stacklevel=3)

    res = _sum_(a)
    if out is not None:
        out[...] = res
        return out
    return res

return _wrapreduction(a, np.add, 'sum', axis, dtype, out, keepdims=keepdims,
                     initial=initial, where=where)

def _any_dispatcher(a, axis=None, out=None, keepdims=None):
    return (a, out)

@array_function_dispatch(_any_dispatcher)
def any(a, axis=None, out=None, keepdims=np._NoValue):
    """
    Test whether any array element along a given axis evaluates to True.
```

```
Returns single boolean unless `axis` is not ``None``

Parameters
-----
a : array_like
    Input array or object that can be converted to an array.
axis : None or int or tuple of ints, optional
    Axis or axes along which a logical OR reduction is performed.
    The default (``axis=None``) is to perform a logical OR over all
    the dimensions of the input array. `axis` may be negative, in
    which case it counts from the last to the first axis.

.. versionadded:: 1.7.0

If this is a tuple of ints, a reduction is performed on multiple
axes, instead of a single axis or all the axes as before.
out : ndarray, optional
    Alternate output array in which to place the result. It must have
    the same shape as the expected output and its type is preserved
    (e.g., if it is of type float, then it will remain so, returning
    1.0 for True and 0.0 for False, regardless of the type of `a`).
    See `ufuncs-output-type` for more details.

keepdims : bool, optional
    If this is set to True, the axes which are reduced are left
    in the result as dimensions with size one. With this option,
    the result will broadcast correctly against the input array.

    If the default value is passed, then `keepdims` will not be
    passed through to the `any` method of sub-classes of
    `ndarray`, however any non-default value will be. If the
    sub-class' method does not implement `keepdims` any
    exceptions will be raised.

Returns
-----
any : bool or ndarray
    A new boolean or `ndarray` is returned unless `out` is specified,
    in which case a reference to `out` is returned.

See Also
-----
ndarray.any : equivalent method

all : Test whether all elements along a given axis evaluate to True.

Notes
-----
Not a Number (NaN), positive infinity and negative infinity evaluate
to `True` because these are not equal to zero.

Examples
-----
>>> np.any([[True, False], [True, True]])
True

>>> np.any([[True, False], [False, False]], axis=0)
array([ True, False])

>>> np.any([-1, 0, 5])
True
```

```
>>> np.any(np.nan)
True

>>> o=np.array(False)
>>> z=np.any([-1, 4, 5], out=o)
>>> z, o
(array(True), array(True))
>>> # Check now that z is a reference to o
>>> z is o
True
>>> id(z), id(o) # identity of z and o           # doctest: +SKIP
(191614240, 191614240)

"""
return _wrapreduction(a, np.logical_or, 'any', axis, None, out, keepdims=keepdims)

def _all_dispatcher(a, axis=None, out=None, keepdims=None):
    return (a, out)

@array_function_dispatch(_all_dispatcher)
def all(a, axis=None, out=None, keepdims=np._NoValue):
    """
    Test whether all array elements along a given axis evaluate to True.

    Parameters
    -----
    a : array_like
        Input array or object that can be converted to an array.
    axis : None or int or tuple of ints, optional
        Axis or axes along which a logical AND reduction is performed.
        The default ('`axis=None``') is to perform a logical AND over all
        the dimensions of the input array. `axis` may be negative, in
        which case it counts from the last to the first axis.

    .. versionadded:: 1.7.0

    If this is a tuple of ints, a reduction is performed on multiple
    axes, instead of a single axis or all the axes as before.
    out : ndarray, optional
        Alternate output array in which to place the result.
        It must have the same shape as the expected output and its
        type is preserved (e.g., if ``dtype(out)`` is float, the result
        will consist of 0.0's and 1.0's). See `ufuncs-output-type` for more
        details.

    keepdims : bool, optional
        If this is set to True, the axes which are reduced are left
        in the result as dimensions with size one. With this option,
        the result will broadcast correctly against the input array.

        If the default value is passed, then `keepdims` will not be
        passed through to the `all` method of sub-classes of
        `ndarray`, however any non-default value will be. If the
        sub-class' method does not implement `keepdims` any
        exceptions will be raised.

    Returns
    -----
    all : ndarray, bool
        A new boolean or array is returned unless `out` is specified,
        in which case a reference to `out` is returned.
```

```
See Also
-----
ndarray.all : equivalent method

any : Test whether any element along a given axis evaluates to True.

Notes
-----
Not a Number (NaN), positive infinity and negative infinity evaluate to `True` because these are not equal to zero.

Examples
-----
>>> np.all([[True, False], [True, True]])
False

>>> np.all([[True, False], [True, True]], axis=0)
array([ True, False])

>>> np.all([-1, 4, 5])
True

>>> np.all([1.0, np.nan])
True

>>> o=np.array(False)
>>> z=np.all([-1, 4, 5], out=o)
>>> id(z), id(o), z
(28293632, 28293632, array(True)) # may vary

"""
return _wrapreduction(a, np.logical_and, 'all', axis, None, out, keepdims=keepdims)

def _cumsum_dispatcher(a, axis=None, dtype=None, out=None):
    return (a, out)

@array_function_dispatch(_cumsum_dispatcher)
def cumsum(a, axis=None, dtype=None, out=None):
    """
    Return the cumulative sum of the elements along a given axis.

    Parameters
    -----
    a : array_like
        Input array.
    axis : int, optional
        Axis along which the cumulative sum is computed. The default
        (None) is to compute the cumsum over the flattened array.
    dtype : dtype, optional
        Type of the returned array and of the accumulator in which the
        elements are summed. If `dtype` is not specified, it defaults
        to the dtype of `a`, unless `a` has an integer dtype with a
        precision less than that of the default platform integer. In
        that case, the default platform integer is used.
    out : ndarray, optional
        Alternative output array in which to place the result. It must
        have the same shape and buffer length as the expected output
        but the type will be cast if necessary. See `ufuncs-output-type` for
        more details.

```

```
Returns
-----
cumsum_along_axis : ndarray.
    A new array holding the result is returned unless `out` is
    specified, in which case a reference to `out` is returned. The
    result has the same size as `a`, and the same shape as `a` if
    `axis` is not None or `a` is a 1-d array.
```

See Also

sum : Sum array elements.

trapz : Integration of array values using the composite trapezoidal rule.

diff : Calculate the n-th discrete difference along given axis.

Notes

Arithmetic is modular when using integer types, and no error is
raised on overflow.

Examples

```
-----
>>> a = np.array([[1,2,3], [4,5,6]])
>>> a
array([[1, 2, 3],
       [4, 5, 6]])
>>> np.cumsum(a)
array([ 1,  3,  6, 10, 15, 21])
>>> np.cumsum(a, dtype=float)      # specifies type of output value(s)
array([ 1.,   3.,   6.,  10.,  15.,  21.])

>>> np.cumsum(a, axis=0)         # sum over rows for each of the 3 columns
array([[1, 2, 3],
       [5, 7, 9]])
>>> np.cumsum(a, axis=1)         # sum over columns for each of the 2 rows
array([[ 1,   3,   6],
       [ 4,   9,  15]])

"""
return _wrapfunc(a, 'cumsum', axis=axis, dtype=dtype, out=out)
```

```
def _ptp_dispatcher(a, axis=None, out=None, keepdims=None):
    return (a, out)
```

```
@array_function_dispatch(_ptp_dispatcher)
def ptp(a, axis=None, out=None, keepdims=np._NoValue):
    """
```

Range of values (maximum - minimum) along an axis.

The name of the function comes from the acronym for 'peak to peak'.

Parameters

```
-----
a : array_like
    Input values.
axis : None or int or tuple of ints, optional
    Axis along which to find the peaks. By default, flatten the
    array. `axis` may be negative, in
    which case it counts from the last to the first axis.
```

```

.. versionadded:: 1.15.0

If this is a tuple of ints, a reduction is performed on multiple
axes, instead of a single axis or all the axes as before.

out : array_like
    Alternative output array in which to place the result. It must
    have the same shape and buffer length as the expected output,
    but the type of the output values will be cast if necessary.

keepdims : bool, optional
    If this is set to True, the axes which are reduced are left
    in the result as dimensions with size one. With this option,
    the result will broadcast correctly against the input array.

    If the default value is passed, then `keepdims` will not be
    passed through to the `ptp` method of sub-classes of
    `ndarray`, however any non-default value will be. If the
    sub-class' method does not implement `keepdims` any
    exceptions will be raised.

Returns
-----
ptp : ndarray
    A new array holding the result, unless `out` was
    specified, in which case a reference to `out` is returned.

Examples
-----
>>> x = np.arange(4).reshape((2,2))
>>> x
array([[0, 1],
       [2, 3]])

>>> np.ptp(x, axis=0)
array([2, 2])

>>> np.ptp(x, axis=1)
array([1, 1])

"""
kwargs = {}
if keepdims is not np._NoValue:
    kwargs['keepdims'] = keepdims
if type(a) is not mu.ndarray:
    try:
        ptp = a.ptp
    except AttributeError:
        pass
    else:
        return ptp(axis=axis, out=out, **kwargs)
return _methods._ptp(a, axis=axis, out=out, **kwargs)

def _amax_dispatcher(a, axis=None, out=None, keepdims=None,
                     initial=None,
                     where=None):
    return (a, out)

@array_function_dispatch(_amax_dispatcher)
def amax(a, axis=None, out=None, keepdims=np._NoValue, initial=np._NoValue,
         where=np._NoValue):
    """

```

Return the maximum of an array or maximum along an axis.

Parameters

a : array_like

Input data.

axis : None or int or tuple of ints, optional

Axis or axes along which to operate. By default, flattened input is used.

.. versionadded:: 1.7.0

If this is a tuple of ints, the maximum is selected over multiple axes, instead of a single axis or all the axes as before.

out : ndarray, optional

Alternative output array in which to place the result. Must be of the same shape and buffer length as the expected output.

See `ufuncs-output-type` for more details.

keepdims : bool, optional

If this is set to True, the axes which are reduced are left in the result as dimensions with size one. With this option, the result will broadcast correctly against the input array.

If the default value is passed, then `keepdims` will not be passed through to the `amax` method of sub-classes of `ndarray`, however any non-default value will be. If the sub-class' method does not implement `keepdims` any exceptions will be raised.

initial : scalar, optional

The minimum value of an output element. Must be present to allow computation on empty slice. See `~numpy.ufunc.reduce` for details.

.. versionadded:: 1.15.0

where : array_like of bool, optional

Elements to compare for the maximum. See `~numpy.ufunc.reduce` for details.

.. versionadded:: 1.17.0

Returns

amax : ndarray or scalar

Maximum of `a`. If `axis` is None, the result is a scalar value.

If `axis` is given, the result is an array of dimension ``a.ndim - 1``.

See Also

amin :

The minimum value of an array along a given axis, propagating any NaNs.

nanmax :

The maximum value of an array along a given axis, ignoring any NaNs.

maximum :

Element-wise maximum of two arrays, propagating any NaNs.

fmax :

Element-wise maximum of two arrays, ignoring any NaNs.

argmax :

Return the indices of the maximum values.

nanmin, minimum, fmin

```
Notes
```

```
-----
```

```
NaN values are propagated, that is if at least one item is NaN, the corresponding max value will be NaN as well. To ignore NaN values (MATLAB behavior), please use nanmax.
```

```
Don't use `amax` for element-wise comparison of 2 arrays; when ``a.shape[0]`` is 2, ``maximum(a[0], a[1])`` is faster than ``amax(a, axis=0)``.
```

```
Examples
```

```
-----
```

```
>>> a = np.arange(4).reshape((2,2))
>>> a
array([[0, 1],
       [2, 3]])
>>> np.amax(a)           # Maximum of the flattened array
3
>>> np.amax(a, axis=0)   # Maxima along the first axis
array([2, 3])
>>> np.amax(a, axis=1)   # Maxima along the second axis
array([1, 3])
>>> np.amax(a, where=[False, True], initial=-1, axis=0)
array([-1, 3])
>>> b = np.arange(5, dtype=float)
>>> b[2] = np.NaN
>>> np.amax(b)
nan
>>> np.amax(b, where=~np.isnan(b), initial=-1)
4.0
>>> np.nanmax(b)
4.0
```

```
You can use an initial value to compute the maximum of an empty slice, or to initialize it to a different value:
```

```
>>> np.max([-50, 10], axis=-1, initial=0)
array([ 0, 10])
```

```
Notice that the initial value is used as one of the elements for which the maximum is determined, unlike for the default argument Python's max function, which is only used for empty iterables.
```

```
>>> np.max([5], initial=6)
6
>>> max([5], default=6)
5
"""
return _wrapreduction(a, np.maximum, 'max', axis, None, out,
                      keepdims=keepdims, initial=initial, where=where)
```

```
def _amin_dispatcher(a, axis=None, out=None, keepdims=None, initial=None,
                     where=None):
    return (a, out)
```

```
@array_function_dispatch(_amin_dispatcher)
def amin(a, axis=None, out=None, keepdims=np._NoValue, initial=np._NoValue,
         where=np._NoValue):
    """

```

```
Return the minimum of an array or minimum along an axis.
```

```
Parameters
-----
a : array_like
    Input data.
axis : None or int or tuple of ints, optional
    Axis or axes along which to operate. By default, flattened input is
    used.

.. versionadded:: 1.7.0

If this is a tuple of ints, the minimum is selected over multiple axes,
instead of a single axis or all the axes as before.
out : ndarray, optional
    Alternative output array in which to place the result. Must
    be of the same shape and buffer length as the expected output.
    See `ufuncs-output-type` for more details.

keepdims : bool, optional
    If this is set to True, the axes which are reduced are left
    in the result as dimensions with size one. With this option,
    the result will broadcast correctly against the input array.

    If the default value is passed, then `keepdims` will not be
    passed through to the `amin` method of sub-classes of
    `ndarray`, however any non-default value will be. If the
    sub-class' method does not implement `keepdims` any
    exceptions will be raised.

initial : scalar, optional
    The maximum value of an output element. Must be present to allow
    computation on empty slice. See `~numpy.ufunc.reduce` for details.

.. versionadded:: 1.15.0

where : array_like of bool, optional
    Elements to compare for the minimum. See `~numpy.ufunc.reduce`
    for details.

.. versionadded:: 1.17.0

Returns
-----
amin : ndarray or scalar
    Minimum of `a`. If `axis` is None, the result is a scalar value.
    If `axis` is given, the result is an array of dimension
    ``a.ndim - 1``.

See Also
-----
amax :
    The maximum value of an array along a given axis, propagating any NaNs.
nanmin :
    The minimum value of an array along a given axis, ignoring any NaNs.
minimum :
    Element-wise minimum of two arrays, propagating any NaNs.
fmin :
    Element-wise minimum of two arrays, ignoring any NaNs.
argmin :
    Return the indices of the minimum values.

nanmax, maximum, fmax
```

Notes

NaN values are propagated, that is if at least one item is NaN, the corresponding min value will be NaN as well. To ignore NaN values (MATLAB behavior), please use nanmin.

Don't use `amin` for element-wise comparison of 2 arrays; when ``a.shape[0]`` is 2, ``minimum(a[0], a[1])`` is faster than ``amin(a, axis=0)``.

Examples

```
>>> a = np.arange(4).reshape((2,2))
>>> a
array([[0, 1],
       [2, 3]])
>>> np.amin(a)           # Minimum of the flattened array
0
>>> np.amin(a, axis=0)   # Minima along the first axis
array([0, 1])
>>> np.amin(a, axis=1)   # Minima along the second axis
array([0, 2])
>>> np.amin(a, where=[False, True], initial=10, axis=0)
array([10,  1])

>>> b = np.arange(5, dtype=float)
>>> b[2] = np.NaN
>>> np.amin(b)
nan
>>> np.amin(b, where=~np.isnan(b), initial=10)
0.0
>>> np.nanmin(b)
0.0

>>> np.min([-50, 10], axis=-1, initial=0)
array([-50,  0])
```

Notice that the initial value is used as one of the elements for which the minimum is determined, unlike for the default argument Python's max function, which is only used for empty iterables.

Notice that this isn't the same as Python's ``default`` argument.

```
>>> np.min([6], initial=5)
5
>>> min([6], default=5)
6
"""
return _wrapreduction(a, np.minimum, 'min', axis, None, out,
                      keepdims=keepdims, initial=initial, where=where)
```

```
def _alen_dispatchcer(a):
    return (a,)
```

```
@array_function_dispatch(_alen_dispatchcer)
def alen(a):
    """
    Return the length of the first dimension of the input array.
```

Parameters

```
a : array_like
    Input array.

Returns
-----
alen : int
    Length of the first dimension of `a`.

See Also
-----
shape, size

Examples
-----
>>> a = np.zeros((7,4,5))
>>> a.shape[0]
7
>>> np.alen(a)
7

"""
# NumPy 1.18.0, 2019-08-02
warnings.warn(
    "`np.alen` is deprecated, use `len` instead",
    DeprecationWarning, stacklevel=2)
try:
    return len(a)
except TypeError:
    return len(array(a, ndmin=1))

def _prod_dispatcher(a, axis=None, dtype=None, out=None, keepdims=None,
                     initial=None, where=None):
    return (a, out)

@array_function_dispatch(_prod_dispatcher)
def prod(a, axis=None, dtype=None, out=None, keepdims=np._NoValue,
         initial=np._NoValue, where=np._NoValue):
    """
    Return the product of array elements over a given axis.

    Parameters
    -----
    a : array_like
        Input data.
    axis : None or int or tuple of ints, optional
        Axis or axes along which a product is performed. The default,
        axis=None, will calculate the product of all the elements in the
        input array. If axis is negative it counts from the last to the
        first axis.

        .. versionadded:: 1.7.0

        If axis is a tuple of ints, a product is performed on all of the
        axes specified in the tuple instead of a single axis or all the
        axes as before.
    dtype : dtype, optional
        The type of the returned array, as well as of the accumulator in
        which the elements are multiplied. The dtype of `a` is used by
        default unless `a` has an integer dtype of less precision than the
        default platform integer. In that case, if `a` is signed then the
        platform integer is used while if `a` is unsigned then an unsigned
```

```
    integer of the same precision as the platform integer is used.
out : ndarray, optional
    Alternative output array in which to place the result. It must have
    the same shape as the expected output, but the type of the output
    values will be cast if necessary.
keepdims : bool, optional
    If this is set to True, the axes which are reduced are left in the
    result as dimensions with size one. With this option, the result
    will broadcast correctly against the input array.

    If the default value is passed, then `keepdims` will not be
    passed through to the `prod` method of sub-classes of
    `ndarray`, however any non-default value will be. If the
    sub-class' method does not implement `keepdims` any
    exceptions will be raised.
initial : scalar, optional
    The starting value for this product. See `~numpy.ufunc.reduce` for details.

.. versionadded:: 1.15.0

where : array_like of bool, optional
    Elements to include in the product. See `~numpy.ufunc.reduce` for details.

.. versionadded:: 1.17.0
```

Returns

```
product_along_axis : ndarray, see `dtype` parameter above.
    An array shaped as `a` but with the specified axis removed.
    Returns a reference to `out` if specified.
```

See Also

```
-----  
ndarray.prod : equivalent method  
ufuncs-output-type
```

Notes

```
-----  
Arithmetic is modular when using integer types, and no error is
raised on overflow. That means that, on a 32-bit platform:
```

```
>>> x = np.array([536870910, 536870910, 536870910, 536870910])
>>> np.prod(x)
16 # may vary
```

The product of an empty array is the neutral element 1:

```
>>> np.prod([])
1.0
```

Examples

```
-----  
By default, calculate the product of all elements:
```

```
>>> np.prod([1.,2.])
2.0
```

Even when the input array is two-dimensional:

```
>>> np.prod([[1.,2.],[3.,4.]])
24.0
```

But we can also specify the axis over which to multiply:

```
>>> np.prod([[1.,2.],[3.,4.]], axis=1)
array([ 2., 12.])

Or select specific elements to include:

>>> np.prod([1., np.nan, 3.], where=[True, False, True])
3.0
```

If the type of `x` is unsigned, then the output type is the unsigned platform integer:

```
>>> x = np.array([1, 2, 3], dtype=np.uint8)
>>> np.prod(x).dtype == np.uint
True
```

If `x` is of a signed integer type, then the output type is the default platform integer:

```
>>> x = np.array([1, 2, 3], dtype=np.int8)
>>> np.prod(x).dtype == int
True
```

You can also start the product with a value other than one:

```
>>> np.prod([1, 2], initial=5)
10
"""
return _wrapreduction(a, np.multiply, 'prod', axis, dtype, out,
                     keepdims=keepdims, initial=initial, where=where)
```

```
def _cumprod_dispatcher(a, axis=None, dtype=None, out=None):
    return (a, out)
```

```
@array_function_dispatch(_cumprod_dispatcher)
def cumprod(a, axis=None, dtype=None, out=None):
    """
    Return the cumulative product of elements along a given axis.
```

Parameters

a : array_like
Input array.

axis : int, optional
Axis along which the cumulative product is computed. By default
the input is flattened.

dtype : dtype, optional

Type of the returned array, as well as of the accumulator in which
the elements are multiplied. If *dtype* is not specified, it
defaults to the dtype of `a`, unless `a` has an integer dtype with
a precision less than that of the default platform integer. In
that case, the default platform integer is used instead.

out : ndarray, optional

Alternative output array in which to place the result. It must
have the same shape and buffer length as the expected output
but the type of the resulting values will be cast if necessary.

Returns

cumprod : ndarray

A new array holding the result is returned unless `out` is

```
specified, in which case a reference to out is returned.
```

See Also

ufuncs-output-type

Notes

Arithmetic is modular when using integer types, and no error is raised on overflow.

Examples

>>> a = np.array([1,2,3])
>>> np.cumprod(a) # intermediate results 1, 1*2
...
total product 1*2*3 = 6
array([1, 2, 6])
>>> a = np.array([[1, 2, 3], [4, 5, 6]])
>>> np.cumprod(a, dtype=float) # specify type of output
array([1., 2., 6., 24., 120., 720.])

The cumulative product for each column (i.e., over the rows) of `a`:

```
>>> np.cumprod(a, axis=0)  
array([[ 1, 2, 3],  
[ 4, 10, 18]])
```

The cumulative product for each row (i.e. over the columns) of `a`:

```
>>> np.cumprod(a, axis=1)  
array([[ 1, 2, 6],  
[ 4, 20, 120]])
```

"""
return _wrapfunc(a, 'cumprod', axis=axis, dtype=dtype, out=out)

```
def _ndim_dispatcher(a):  
    return (a,
```

```
@array_function_dispatch(_ndim_dispatcher)  
def ndim(a):  
    """  
    Return the number of dimensions of an array.
```

Parameters

a : array_like
Input array. If it is not already an ndarray, a conversion is attempted.

Returns

number_of_dimensions : int
The number of dimensions in `a`. Scalars are zero-dimensional.

See Also

ndarray.ndim : equivalent method
shape : dimensions of array
ndarray.shape : dimensions of array

```
Examples
-----
>>> np.ndim([[1,2,3],[4,5,6]])
2
>>> np.ndim(np.array([[1,2,3],[4,5,6]]))
2
>>> np.ndim(1)
0

"""
try:
    return a.ndim
except AttributeError:
    return asarray(a).ndim

def _size_dispatcher(a, axis=None):
    return (a,)

@array_function_dispatch(_size_dispatcher)
def size(a, axis=None):
    """
    Return the number of elements along a given axis.

    Parameters
    -----
    a : array_like
        Input data.
    axis : int, optional
        Axis along which the elements are counted. By default, give
        the total number of elements.

    Returns
    -----
    element_count : int
        Number of elements along the specified axis.

    See Also
    -----
    shape : dimensions of array
    ndarray.shape : dimensions of array
    ndarray.size : number of elements in array

Examples
-----
>>> a = np.array([[1,2,3],[4,5,6]])
>>> np.size(a)
6
>>> np.size(a,1)
3
>>> np.size(a,0)
2

"""
if axis is None:
    try:
        return a.size
    except AttributeError:
        return asarray(a).size
else:
    try:
        return a.shape[axis]
```

```
        except AttributeError:
            return asarray(a).shape[axis]

def _around_dispatcher(a, decimals=None, out=None):
    return (a, out)

@array_function_dispatch(_around_dispatcher)
def around(a, decimals=0, out=None):
    """
    Evenly round to the given number of decimals.

    Parameters
    -----
    a : array_like
        Input data.
    decimals : int, optional
        Number of decimal places to round to (default: 0). If
        decimals is negative, it specifies the number of positions to
        the left of the decimal point.
    out : ndarray, optional
        Alternative output array in which to place the result. It must have
        the same shape as the expected output, but the type of the output
        values will be cast if necessary. See `ufuncs-output-type` for more
        details.

    Returns
    -----
    rounded_array : ndarray
        An array of the same type as `a`, containing the rounded values.
        Unless `out` was specified, a new array is created. A reference to
        the result is returned.

    The real and imaginary parts of complex numbers are rounded
    separately. The result of rounding a float is a float.

    See Also
    -----
    ndarray.round : equivalent method

    ceil, fix, floor, rint, trunc
```

Notes

For values exactly halfway between rounded decimal values, NumPy rounds to the nearest even value. Thus 1.5 and 2.5 round to 2.0, -0.5 and 0.5 round to 0.0, etc.

``np.around`` uses a fast but sometimes inexact algorithm to round floating-point datatypes. For positive `decimals` it is equivalent to ``np.true_divide(np.rint(a * 10**decimals), 10**decimals)``, which has error due to the inexact representation of decimal fractions in the IEEE floating point standard [1]_ and errors introduced when scaling by powers of ten. For instance, note the extra "1" in the following:

```
>>> np.round(56294995342131.5, 3)
56294995342131.51
```

If your goal is to print such values with a fixed number of decimals, it is preferable to use numpy's float printing routines to limit the number of printed decimals:

```
>>> np.format_float_positional(56294995342131.5, precision=3)
'56294995342131.5'
```

The float printing routines use an accurate but much more computationally demanding algorithm to compute the number of digits after the decimal point.

Alternatively, Python's builtin `round` function uses a more accurate but slower algorithm for 64-bit floating point values:

```
>>> round(56294995342131.5, 3)
56294995342131.5
>>> np.round(16.055, 2), round(16.055, 2) # equals 16.0549999999999997
(16.06, 16.05)
```

References

- .. [1] "Lecture Notes on the Status of IEEE 754", William Kahan,
<https://people.eecs.berkeley.edu/~wkahan/ieee754status/IEEE754.PDF>
- .. [2] "How Futile are Mindless Assessments of
Roundoff in Floating-Point Computation?", William Kahan,
<https://people.eecs.berkeley.edu/~wkahan/Mindless.pdf>

Examples

```
>>> np.around([0.37, 1.64])
array([0., 2.])
>>> np.around([0.37, 1.64], decimals=1)
array([0.4, 1.6])
>>> np.around([.5, 1.5, 2.5, 3.5, 4.5]) # rounds to nearest even value
array([0., 2., 2., 4., 4.])
>>> np.around([1,2,3,11], decimals=1) # ndarray of ints is returned
array([1, 2, 3, 11])
>>> np.around([1,2,3,11], decimals=-1)
array([0, 0, 0, 10])

"""
return _wrapfunc(a, 'round', decimals=decimals, out=out)
```

```
def _mean_dispatcher(a, axis=None, dtype=None, out=None, keepdims=None):
    return (a, out)
```

```
@array_function_dispatch(_mean_dispatcher)
def mean(a, axis=None, dtype=None, out=None, keepdims=np._NoValue):
    """
    Compute the arithmetic mean along the specified axis.
```

Returns the average of the array elements. The average is taken over the flattened array by default, otherwise over the specified axis. `float64` intermediate and return values are used for integer inputs.

Parameters

```
a : array_like
    Array containing numbers whose mean is desired. If `a` is not an
    array, a conversion is attempted.
axis : None or int or tuple of ints, optional
    Axis or axes along which the means are computed. The default is to
    compute the mean of the flattened array.
```

```
.. versionadded:: 1.7.0

If this is a tuple of ints, a mean is performed over multiple axes,
instead of a single axis or all the axes as before.

dtype : data-type, optional
    Type to use in computing the mean. For integer inputs, the default
    is `float64`; for floating point inputs, it is the same as the
    input dtype.

out : ndarray, optional
    Alternate output array in which to place the result. The default
    is ``None``; if provided, it must have the same shape as the
    expected output, but the type will be cast if necessary.
    See `ufuncs-output-type` for more details.
```

keepdims : bool, optional
If this is set to True, the axes which are reduced are left
in the result as dimensions with size one. With this option,
the result will broadcast correctly against the input array.

If the default value is passed, then `keepdims` will not be
passed through to the `mean` method of sub-classes of
`ndarray`, however any non-default value will be. If the
sub-class' method does not implement `keepdims` any
exceptions will be raised.

Returns

m : ndarray, see dtype parameter above
 If `out=None`, returns a new array containing the mean values,
 otherwise a reference to the output array is returned.

See Also

average : Weighted average
std, var, nanmean, nanstd, nanvar

Notes

The arithmetic mean is the sum of the elements along the axis divided
by the number of elements.

Note that for floating-point input, the mean is computed using the
same precision the input has. Depending on the input data, this can
cause the results to be inaccurate, especially for `float32` (see
example below). Specifying a higher-precision accumulator using the
`dtype` keyword can alleviate this issue.

By default, `float16` results are computed using `float32` intermediates
for extra precision.

Examples

>>> a = np.array([[1, 2], [3, 4]])
>>> np.mean(a)
2.5
>>> np.mean(a, axis=0)
array([2., 3.])
>>> np.mean(a, axis=1)
array([1.5, 3.5])

In single precision, `mean` can be inaccurate:

```

>>> a = np.zeros((2, 512*512), dtype=np.float32)
>>> a[0, :] = 1.0
>>> a[1, :] = 0.1
>>> np.mean(a)
0.54999924

Computing the mean in float64 is more accurate:

>>> np.mean(a, dtype=np.float64)
0.55000000074505806 # may vary

"""
kwargs = {}
if keepdims is not np._NoValue:
    kwargs['keepdims'] = keepdims
if type(a) is not mu.ndarray:
    try:
        mean = a.mean
    except AttributeError:
        pass
    else:
        return mean(axis=axis, dtype=dtype, out=out, **kwargs)

return _methods._mean(a, axis=axis, dtype=dtype,
                      out=out, **kwargs)

def _std_dispatcher(
    a, axis=None, dtype=None, out=None, ddof=None, keepdims=None):
    return (a, out)

@array_function_dispatch(_std_dispatcher)
def std(a, axis=None, dtype=None, out=None, ddof=0, keepdims=np._NoValue):
    """
    Compute the standard deviation along the specified axis.

    Returns the standard deviation, a measure of the spread of a distribution,
    of the array elements. The standard deviation is computed for the
    flattened array by default, otherwise over the specified axis.

    Parameters
    -----
    a : array_like
        Calculate the standard deviation of these values.
    axis : None or int or tuple of ints, optional
        Axis or axes along which the standard deviation is computed. The
        default is to compute the standard deviation of the flattened array.

        .. versionadded:: 1.7.0

        If this is a tuple of ints, a standard deviation is performed over
        multiple axes, instead of a single axis or all the axes as before.

    dtype : dtype, optional
        Type to use in computing the standard deviation. For arrays of
        integer type the default is float64, for arrays of float types it is
        the same as the array type.

    out : ndarray, optional
        Alternative output array in which to place the result. It must have
        the same shape as the expected output but the type (of the calculated
        values) will be cast if necessary.

    ddof : int, optional
        Means Delta Degrees of Freedom. The divisor used in calculations

```

```
is ``N - ddof``, where ``N`` represents the number of elements.  
By default `ddof` is zero.  
keepdims : bool, optional  
    If this is set to True, the axes which are reduced are left  
    in the result as dimensions with size one. With this option,  
    the result will broadcast correctly against the input array.  
  
    If the default value is passed, then `keepdims` will not be  
    passed through to the `std` method of sub-classes of  
    `ndarray`, however any non-default value will be. If the  
    sub-class' method does not implement `keepdims` any  
    exceptions will be raised.
```

Returns

```
-----  
standard_deviation : ndarray, see dtype parameter above.  
    If `out` is None, return a new array containing the standard deviation,  
    otherwise return a reference to the output array.
```

See Also

```
-----  
var, mean, nanmean, nanstd, nanvar  
ufuncs-output-type
```

Notes

```
-----  
The standard deviation is the square root of the average of the squared  
deviations from the mean, i.e., ``std = sqrt(mean(abs(x - x.mean())**2))``.
```

The average squared deviation is normally calculated as
``x.sum() / N``, where ``N = len(x)``. If, however, `ddof` is specified,
the divisor ``N - ddof`` is used instead. In standard statistical
practice, ``ddof=1`` provides an unbiased estimator of the variance
of the infinite population. ``ddof=0`` provides a maximum likelihood
estimate of the variance for normally distributed variables. The
standard deviation computed in this function is the square root of
the estimated variance, so even with ``ddof=1``, it will not be an
unbiased estimate of the standard deviation per se.

Note that, for complex numbers, `std` takes the absolute
value before squaring, so that the result is always real and nonnegative.

For floating-point input, the *std* is computed using the same
precision the input has. Depending on the input data, this can cause
the results to be inaccurate, especially for float32 (see example below).
Specifying a higher-accuracy accumulator using the `dtype` keyword can
alleviate this issue.

Examples

```
-----  
>>> a = np.array([[1, 2], [3, 4]])  
>>> np.std(a)  
1.1180339887498949 # may vary  
>>> np.std(a, axis=0)  
array([1., 1.])  
>>> np.std(a, axis=1)  
array([0.5, 0.5])
```

In single precision, std() can be inaccurate:

```
>>> a = np.zeros((2, 512*512), dtype=np.float32)  
>>> a[0, :] = 1.0  
>>> a[1, :] = 0.1
```

```

>>> np.std(a)
0.45000005

Computing the standard deviation in float64 is more accurate:

>>> np.std(a, dtype=np.float64)
0.4499999925494177 # may vary

"""
kwargs = {}
if keepdims is not np._NoValue:
    kwargs['keepdims'] = keepdims

if type(a) is not mu.ndarray:
    try:
        std = a.std
    except AttributeError:
        pass
    else:
        return std(axis=axis, dtype=dtype, out=out, ddof=ddof, **kwargs)

return _methods._std(a, axis=axis, dtype=dtype, out=out, ddof=ddof,
                     **kwargs)

def _var_dispatcher(
    a, axis=None, dtype=None, out=None, ddof=None, keepdims=None):
    return (a, out)

@array_function_dispatch(_var_dispatcher)
def var(a, axis=None, dtype=None, out=None, ddof=0, keepdims=np._NoValue):
    """
    Compute the variance along the specified axis.

    Returns the variance of the array elements, a measure of the spread of a
    distribution. The variance is computed for the flattened array by
    default, otherwise over the specified axis.

    Parameters
    -----
    a : array_like
        Array containing numbers whose variance is desired. If `a` is not an
        array, a conversion is attempted.
    axis : None or int or tuple of ints, optional
        Axis or axes along which the variance is computed. The default is to
        compute the variance of the flattened array.

    .. versionadded:: 1.7.0

    If this is a tuple of ints, a variance is performed over multiple axes,
    instead of a single axis or all the axes as before.
    dtype : data-type, optional
        Type to use in computing the variance. For arrays of integer type
        the default is `float64`; for arrays of float types it is the same as
        the array type.
    out : ndarray, optional
        Alternate output array in which to place the result. It must have
        the same shape as the expected output, but the type is cast if
        necessary.
    ddof : int, optional
        "Delta Degrees of Freedom": the divisor used in the calculation is
        ``N - ddof``, where ``N`` represents the number of elements. By

```

```
    default `ddof` is zero.  
keepdims : bool, optional  
    If this is set to True, the axes which are reduced are left  
    in the result as dimensions with size one. With this option,  
    the result will broadcast correctly against the input array.
```

If the default value is passed, then `keepdims` will not be passed through to the `var` method of sub-classes of `ndarray`, however any non-default value will be. If the sub-class' method does not implement `keepdims` any exceptions will be raised.

Returns

```
-----  
variance : ndarray, see dtype parameter above  
    If ``out=None``, returns a new array containing the variance;  
    otherwise, a reference to the output array is returned.
```

See Also

```
-----  
std, mean, nanmean, nanstd, nanvar  
ufuncs-output-type
```

Notes

The variance is the average of the squared deviations from the mean, i.e., ``var = mean(abs(x - x.mean())**2)``.

The mean is normally calculated as ``x.sum() / N``, where ``N = len(x)``. If, however, `ddof` is specified, the divisor ``N - ddof`` is used instead. In standard statistical practice, ``ddof=1`` provides an unbiased estimator of the variance of a hypothetical infinite population. ``ddof=0`` provides a maximum likelihood estimate of the variance for normally distributed variables.

Note that for complex numbers, the absolute value is taken before squaring, so that the result is always real and nonnegative.

For floating-point input, the variance is computed using the same precision the input has. Depending on the input data, this can cause the results to be inaccurate, especially for `float32` (see example below). Specifying a higher-accuracy accumulator using the ``dtype`` keyword can alleviate this issue.

Examples

```
-----  
>>> a = np.array([[1, 2], [3, 4]])  
>>> np.var(a)  
1.25  
>>> np.var(a, axis=0)  
array([1., 1.])  
>>> np.var(a, axis=1)  
array([0.25, 0.25])
```

In single precision, var() can be inaccurate:

```
>>> a = np.zeros((2, 512*512), dtype=np.float32)  
>>> a[0, :] = 1.0  
>>> a[1, :] = 0.1  
>>> np.var(a)  
0.20250003
```

Computing the variance in float64 is more accurate:

```
>>> np.var(a, dtype=np.float64)
0.20249999932944759 # may vary
>>> ((1-0.55)**2 + (0.1-0.55)**2)/2
0.2025

"""
kwargs = {}
if keepdims is not np._NoValue:
    kwargs['keepdims'] = keepdims

if type(a) is not mu.ndarray:
    try:
        var = a.var

    except AttributeError:
        pass
    else:
        return var(axis=axis, dtype=dtype, out=out, ddof=ddof, **kwargs)

return _methods._var(a, axis=axis, dtype=dtype, out=out, ddof=ddof,
                     **kwargs)

# Aliases of other functions. These have their own definitions only so that
# they can have unique docstrings.

@array_function_dispatch(_around_dispatcher)
def round_(a, decimals=0, out=None):
    """
    Round an array to the given number of decimals.

    See Also
    -----
    around : equivalent function; see for details.
    """
    return around(a, decimals=decimals, out=out)

@array_function_dispatch(_prod_dispatcher, verify=False)
def product(*args, **kwargs):
    """
    Return the product of array elements over a given axis.

    See Also
    -----
    prod : equivalent function; see for details.
    """
    return prod(*args, **kwargs)

@array_function_dispatch(_cumprod_dispatcher, verify=False)
def cumproduct(*args, **kwargs):
    """
    Return the cumulative product over the given axis.

    See Also
    -----
    cumprod : equivalent function; see for details.
    """
    return cumprod(*args, **kwargs)
```

```
@array_function_dispatch(_any_dispatcher, verify=False)
def sometrue(*args, **kwargs):
    """
    Check whether some values are true.

    Refer to `any` for full documentation.

    See Also
    -----
    any : equivalent function; see for details.
    """
    return any(*args, **kwargs)

@array_function_dispatch(_all_dispatcher, verify=False)
def alltrue(*args, **kwargs):
    """
    Check if all elements of input array are true.

    See Also
    -----
    numpy.all : Equivalent function; see for details.
    """
    return all(*args, **kwargs)
```

https://github.com/scikit-learn/scikit-learn/blob/master/sklearn/model_selection/_split.py

```
"""
The :mod:`sklearn.model_selection._split` module includes classes and
functions to split the data based on a preset strategy.
"""

# Author: Alexandre Gramfort <alexandre.gramfort@inria.fr>,
#         Gael Varoquaux <gael.varoquaux@normalesup.org>,
#         Olivier Grisel <olivier.grisel@ensta.org>
#         Raghav RV <rvrاغhav93@gmail.com>
# License: BSD 3 clause

from collections.abc import Iterable
import warnings
from itertools import chain, combinations
from math import ceil, floor
import numbers
from abc import ABCMeta, abstractmethod
from inspect import signature

import numpy as np
from scipy.special import comb

from ..utils import indexable, check_random_state, _safe_indexing
from ..utils import _approximate_mode
from ..utils.validation import _num_samples, column_or_1d
from ..utils.validation import check_array
from ..utils.validation import _deprecate_positional_args
from ..utils.multiclass import type_of_target
from ..base import _pprint

__all__ = ['BaseCrossValidator',
           'KFold',
           'GroupKFold',
           'LeaveOneGroupOut',
           'LeaveOneOut',
           'LeavePGroupsOut',
           'LeavePOut',
           'RepeatedStratifiedKFold',
           'RepeatedKFold',
           'ShuffleSplit',
           'GroupShuffleSplit',
           'StratifiedKFold',
           'StratifiedShuffleSplit',
           'PredefinedSplit',
           'train_test_split',
           'check_cv']

class BaseCrossValidator(metaclass=ABCMeta):
    """Base class for all cross-validation

    Implementations must define `'_iter_test_masks` or `'_iter_test_indices`.

    """
    def split(self, X, y=None, groups=None):
        """Generate indices to split data into training and test set.
```

```

Parameters
-----
X : array-like of shape (n_samples, n_features)
    Training data, where n_samples is the number of samples
    and n_features is the number of features.

y : array-like of shape (n_samples,)
    The target variable for supervised learning problems.

groups : array-like of shape (n_samples,), default=None
    Group labels for the samples used while splitting the dataset into
    train/test set.

Yields
-----
train : ndarray
    The training set indices for that split.

test : ndarray
    The testing set indices for that split.
"""
X, y, groups = indexable(X, y, groups)
indices = np.arange(_num_samples(X))
for test_index in self._iter_test_masks(X, y, groups):
    train_index = indices[np.logical_not(test_index)]
    test_index = indices[test_index]
    yield train_index, test_index

# Since subclasses must implement either _iter_test_masks or
# _iter_test_indices, neither can be abstract.
def _iter_test_masks(self, X=None, y=None, groups=None):
    """Generates boolean masks corresponding to test sets.

    By default, delegates to _iter_test_indices(X, y, groups)
    """
    for test_index in self._iter_test_indices(X, y, groups):
        test_mask = np.zeros(_num_samples(X), dtype=np.bool)
        test_mask[test_index] = True
        yield test_mask

def _iter_test_indices(self, X=None, y=None, groups=None):
    """Generates integer indices corresponding to test sets."""
    raise NotImplementedError

@abstractmethod
def get_n_splits(self, X=None, y=None, groups=None):
    """Returns the number of splitting iterations in the cross-validator"""

def __repr__(self):
    return _build_repr(self)

class LeaveOneOut(BaseCrossValidator):
    """Leave-One-Out cross-validator

    Provides train/test indices to split data in train/test sets. Each
    sample is used once as a test set (singleton) while the remaining
    samples form the training set.

    Note: ``LeaveOneOut()`` is equivalent to ``KFold(n_splits=n)`` and
    ``LeavePOut(p=1)`` where ``n`` is the number of samples.

```

Due to the high number of test sets (which is the same as the number of samples) this cross-validation method can be very costly. For large datasets one should favor :class:`KFold`, :class:`ShuffleSplit` or :class:`StratifiedKFold`.

Read more in the :ref:`User Guide <cross_validation>`.

Examples

```
>>> import numpy as np
>>> from sklearn.model_selection import LeaveOneOut
>>> X = np.array([[1, 2], [3, 4]])
>>> y = np.array([1, 2])
>>> loo = LeaveOneOut()
>>> loo.get_n_splits(X)
2
>>> print(loo)
LeaveOneOut()
>>> for train_index, test_index in loo.split(X):
...     print("TRAIN:", train_index, "TEST:", test_index)
...     X_train, X_test = X[train_index], X[test_index]
...     y_train, y_test = y[train_index], y[test_index]
...     print(X_train, X_test, y_train, y_test)
TRAIN: [1] TEST: [0]
[[3 4]] [[1 2]] [2] [1]
TRAIN: [0] TEST: [1]
[[1 2]] [[3 4]] [1] [2]
```

See also

LeaveOneGroupOut

For splitting the data according to explicit, domain-specific stratification of the dataset.

GroupKFold: K-fold iterator variant with non-overlapping groups.

```
""""
def _iter_test_indices(self, X, y=None, groups=None):
    n_samples = _num_samples(X)
    if n_samples <= 1:
        raise ValueError(
            'Cannot perform LeaveOneOut with n_samples={}'.format(
                n_samples))
    return range(n_samples)

def get_n_splits(self, X, y=None, groups=None):
    """Returns the number of splitting iterations in the cross-validator
```

Parameters

X : array-like of shape (n_samples, n_features)
Training data, where n_samples is the number of samples
and n_features is the number of features.

y : object
Always ignored, exists for compatibility.

groups : object
Always ignored, exists for compatibility.

Returns

```

n_splits : int
    Returns the number of splitting iterations in the cross-validator.
"""
if X is None:
    raise ValueError("The 'X' parameter should not be None.")
return _num_samples(X)

class LeavePOut(BaseCrossValidator):
    """Leave-P-Out cross-validator

    Provides train/test indices to split data in train/test sets. This results
    in testing on all distinct samples of size p, while the remaining n - p
    samples form the training set in each iteration.

    Note: ``LeavePOut(p)`` is NOT equivalent to
    ``KFold(n_splits=n_samples // p)`` which creates non-overlapping test sets.

    Due to the high number of iterations which grows combinatorically with the
    number of samples this cross-validation method can be very costly. For
    large datasets one should favor :class:`KFold`, :class:`StratifiedKFold`
    or :class:`ShuffleSplit`.

    Read more in the :ref:`User Guide <cross_validation>`.

    Parameters
    -----
    p : int
        Size of the test sets. Must be strictly less than the number of
        samples.

    Examples
    -----
    >>> import numpy as np
    >>> from sklearn.model_selection import LeavePOut
    >>> X = np.array([[1, 2], [3, 4], [5, 6], [7, 8]])
    >>> y = np.array([1, 2, 3, 4])
    >>> lpo = LeavePOut(2)
    >>> lpo.get_n_splits(X)
    6
    >>> print(lpo)
    LeavePOut(p=2)
    >>> for train_index, test_index in lpo.split(X):
    ...     print("TRAIN:", train_index, "TEST:", test_index)
    ...     X_train, X_test = X[train_index], X[test_index]
    ...     y_train, y_test = y[train_index], y[test_index]
    TRAIN: [2 3] TEST: [0 1]
    TRAIN: [1 3] TEST: [0 2]
    TRAIN: [1 2] TEST: [0 3]
    TRAIN: [0 3] TEST: [1 2]
    TRAIN: [0 2] TEST: [1 3]
    TRAIN: [0 1] TEST: [2 3]
"""

def __init__(self, p):
    self.p = p

def _iter_test_indices(self, X, y=None, groups=None):
    n_samples = _num_samples(X)
    if n_samples <= self.p:
        raise ValueError(
            'p={} must be strictly less than the number of '
            'samples={}'.format(self.p, n_samples))

```

```

        )
    for combination in combinations(range(n_samples), self.p):
        yield np.array(combination)

def get_n_splits(self, X, y=None, groups=None):
    """Returns the number of splitting iterations in the cross-validator

    Parameters
    -----
    X : array-like of shape (n_samples, n_features)
        Training data, where n_samples is the number of samples
        and n_features is the number of features.

    y : object
        Always ignored, exists for compatibility.

    groups : object
        Always ignored, exists for compatibility.
    """
    if X is None:
        raise ValueError("The 'X' parameter should not be None.")
    return int(comb(_num_samples(X), self.p, exact=True))

class _BaseKFold(BaseCrossValidator, metaclass=ABCMeta):
    """Base class for KFold, GroupKFold, and StratifiedKFold"""

    @abstractmethod
    @_deprecate_positional_args
    def __init__(self, n_splits, *, shuffle, random_state):
        if not isinstance(n_splits, numbers.Integral):
            raise ValueError('The number of folds must be of Integral type. '
                             '%s of type %s was passed.' %
                             (n_splits, type(n_splits)))
        n_splits = int(n_splits)

        if n_splits <= 1:
            raise ValueError(
                "k-fold cross-validation requires at least one"
                " train/test split by setting n_splits=2 or more,"
                " got n_splits={0}.".format(n_splits))

        if not isinstance(shuffle, bool):
            raise TypeError("shuffle must be True or False;"
                            " got {0}.".format(shuffle))

        if not shuffle and random_state is not None: # None is the default
            # TODO 0.24: raise a ValueError instead of a warning
            warnings.warn(
                'Setting a random_state has no effect since shuffle is '
                "'False. This will raise an error in 0.24. You should leave "
                "'random_state to its default (None), or set shuffle=True.'",
                FutureWarning
            )

        self.n_splits = n_splits
        self.shuffle = shuffle
        self.random_state = random_state

    def split(self, X, y=None, groups=None):
        """Generate indices to split data into training and test set.

        Parameters

```

```

-----
X : array-like of shape (n_samples, n_features)
    Training data, where n_samples is the number of samples
    and n_features is the number of features.

y : array-like of shape (n_samples,), default=None
    The target variable for supervised learning problems.

groups : array-like of shape (n_samples,), default=None
    Group labels for the samples used while splitting the dataset into
    train/test set.

Yields
-----
train : ndarray
    The training set indices for that split.

test : ndarray
    The testing set indices for that split.
"""
X, y, groups = indexable(X, y, groups)
n_samples = _num_samples(X)
if self.n_splits > n_samples:
    raise ValueError(
        ("Cannot have number of splits n_splits={0} greater"
         " than the number of samples: n_samples={1}.")
        .format(self.n_splits, n_samples))

for train, test in super().split(X, y, groups):
    yield train, test

def get_n_splits(self, X=None, y=None, groups=None):
    """Returns the number of splitting iterations in the cross-validator

Parameters
-----
X : object
    Always ignored, exists for compatibility.

y : object
    Always ignored, exists for compatibility.

groups : object
    Always ignored, exists for compatibility.

Returns
-----
n_splits : int
    Returns the number of splitting iterations in the cross-validator.
"""
    return self.n_splits

class KFold(_BaseKFold):
    """K-Folds cross-validator

Provides train/test indices to split data in train/test sets. Split
dataset into k consecutive folds (without shuffling by default).

Each fold is then used once as a validation while the k - 1 remaining
folds form the training set.

Read more in the :ref:`User Guide <cross_validation>`.

```

Parameters

n_splits : int, default=5
Number of folds. Must be at least 2.

.. versionchanged:: 0.22
``n_splits`` default value changed from 3 to 5.

shuffle : bool, default=False
Whether to shuffle the data before splitting into batches.
Note that the samples within each split will not be shuffled.

random_state : int or RandomState instance, default=None
When `shuffle` is True, `random_state` affects the ordering of the indices, which controls the randomness of each fold. Otherwise, this parameter has no effect.
Pass an int for reproducible output across multiple function calls.
See :term:`Glossary <random_state>`.

Examples

```
>>> import numpy as np
>>> from sklearn.model_selection import KFold
>>> X = np.array([[1, 2], [3, 4], [1, 2], [3, 4]])
>>> y = np.array([1, 2, 3, 4])
>>> kf = KFold(n_splits=2)
>>> kf.get_n_splits(X)
2
>>> print(kf)
KFold(n_splits=2, random_state=None, shuffle=False)
>>> for train_index, test_index in kf.split(X):
...     print("TRAIN:", train_index, "TEST:", test_index)
...     X_train, X_test = X[train_index], X[test_index]
...     y_train, y_test = y[train_index], y[test_index]
TRAIN: [2 3] TEST: [0 1]
TRAIN: [0 1] TEST: [2 3]
```

Notes

The first `` $n_{samples} \% n_{splits}$ `` folds have size `` $n_{samples} // n_{splits} + 1$ ``, other folds have size `` $n_{samples} // n_{splits}$ ``, where `` $n_{samples}$ `` is the number of samples.

Randomized CV splitters may return different results for each call of split. You can make the results identical by setting `random_state` to an integer.

See also

StratifiedKFold

Takes group information into account to avoid building folds with imbalanced class distributions (for binary or multiclass classification tasks).

GroupKFold: K-fold iterator variant with non-overlapping groups.

RepeatedKFold: Repeats K-Fold n times.

"""

```
@_deprecate_positional_args
def __init__(self, n_splits=5, *, shuffle=False,
             random_state=None):
    super().__init__(n_splits=n_splits, shuffle=shuffle,
```

```

        random_state=random_state)

def _iter_test_indices(self, X, y=None, groups=None):
    n_samples = _num_samples(X)
    indices = np.arange(n_samples)
    if self.shuffle:
        check_random_state(self.random_state).shuffle(indices)

    n_splits = self.n_splits
    fold_sizes = np.full(n_splits, n_samples // n_splits, dtype=np.int)
    fold_sizes[:n_samples % n_splits] += 1
    current = 0
    for fold_size in fold_sizes:
        start, stop = current, current + fold_size
        yield indices[start:stop]
        current = stop

class GroupKFold(_BaseKFold):
    """K-fold iterator variant with non-overlapping groups.

    The same group will not appear in two different folds (the number of
    distinct groups has to be at least equal to the number of folds).

    The folds are approximately balanced in the sense that the number of
    distinct groups is approximately the same in each fold.

    Parameters
    -----
    n_splits : int, default=5
        Number of folds. Must be at least 2.

    .. versionchanged:: 0.22
        ``n_splits`` default value changed from 3 to 5.

    Examples
    -----
    >>> import numpy as np
    >>> from sklearn.model_selection import GroupKFold
    >>> X = np.array([[1, 2], [3, 4], [5, 6], [7, 8]])
    >>> y = np.array([1, 2, 3, 4])
    >>> groups = np.array([0, 0, 2, 2])
    >>> group_kfold = GroupKFold(n_splits=2)
    >>> group_kfold.get_n_splits(X, y, groups)
    2
    >>> print(group_kfold)
    GroupKFold(n_splits=2)
    >>> for train_index, test_index in group_kfold.split(X, y, groups):
    ...     print("TRAIN:", train_index, "TEST:", test_index)
    ...     X_train, X_test = X[train_index], X[test_index]
    ...     y_train, y_test = y[train_index], y[test_index]
    ...     print(X_train, X_test, y_train, y_test)
    ...
    TRAIN: [0 1] TEST: [2 3]
    [[1 2]
     [3 4]] [[5 6]
     [7 8]] [1 2] [3 4]
    TRAIN: [2 3] TEST: [0 1]
    [[5 6]
     [7 8]] [[1 2]
     [3 4]] [3 4] [1 2]

```

See also

```

-----
LeaveOneGroupOut
    For splitting the data according to explicit domain-specific
    stratification of the dataset.
"""
def __init__(self, n_splits=5):
    super().__init__(n_splits, shuffle=False, random_state=None)

def _iter_test_indices(self, X, y, groups):
    if groups is None:
        raise ValueError("The 'groups' parameter should not be None.")
    groups = check_array(groups, ensure_2d=False, dtype=None)

    unique_groups, groups = np.unique(groups, return_inverse=True)
    n_groups = len(unique_groups)

    if self.n_splits > n_groups:
        raise ValueError("Cannot have number of splits n_splits=%d greater"
                         " than the number of groups: %d."
                         % (self.n_splits, n_groups))

    # Weight groups by their number of occurrences
    n_samples_per_group = np.bincount(groups)

    # Distribute the most frequent groups first
    indices = np.argsort(n_samples_per_group)[::-1]
    n_samples_per_group = n_samples_per_group[indices]

    # Total weight of each fold
    n_samples_per_fold = np.zeros(self.n_splits)

    # Mapping from group index to fold index
    group_to_fold = np.zeros(len(unique_groups))

    # Distribute samples by adding the largest weight to the lightest fold
    for group_index, weight in enumerate(n_samples_per_group):
        lightest_fold = np.argmin(n_samples_per_fold)
        n_samples_per_fold[lightest_fold] += weight
        group_to_fold[indices[group_index]] = lightest_fold

    indices = group_to_fold[groups]

    for f in range(self.n_splits):
        yield np.where(indices == f)[0]

def split(self, X, y=None, groups=None):
    """Generate indices to split data into training and test set.

    Parameters
    -----
    X : array-like of shape (n_samples, n_features)
        Training data, where n_samples is the number of samples
        and n_features is the number of features.

    y : array-like of shape (n_samples,), default=None
        The target variable for supervised learning problems.

    groups : array-like of shape (n_samples,)
        Group labels for the samples used while splitting the dataset into
        train/test set.

    Yields
    -----

```

```

train : ndarray
    The training set indices for that split.

test : ndarray
    The testing set indices for that split.
"""
    return super().split(X, y, groups)

class StratifiedKFold(_BaseKFold):
    """Stratified K-Folds cross-validator

    Provides train/test indices to split data in train/test sets.

    This cross-validation object is a variation of KFold that returns
    stratified folds. The folds are made by preserving the percentage of
    samples for each class.

    Read more in the :ref:`User Guide <cross_validation>`.

    Parameters
    -----
    n_splits : int, default=5
        Number of folds. Must be at least 2.

        .. versionchanged:: 0.22
            ``n_splits`` default value changed from 3 to 5.

    shuffle : bool, default=False
        Whether to shuffle each class's samples before splitting into batches.
        Note that the samples within each split will not be shuffled.

    random_state : int or RandomState instance, default=None
        When `shuffle` is True, `random_state` affects the ordering of the
        indices, which controls the randomness of each fold for each class.
        Otherwise, leave `random_state` as `None`.
        Pass an int for reproducible output across multiple function calls.
        See :term:`Glossary <random_state>`.

    Examples
    -----
    >>> import numpy as np
    >>> from sklearn.model_selection import StratifiedKFold
    >>> X = np.array([[1, 2], [3, 4], [1, 2], [3, 4]])
    >>> y = np.array([0, 0, 1, 1])
    >>> skf = StratifiedKFold(n_splits=2)
    >>> skf.get_n_splits(X, y)
    2
    >>> print(skf)
    StratifiedKFold(n_splits=2, random_state=None, shuffle=False)
    >>> for train_index, test_index in skf.split(X, y):
    ...     print("TRAIN:", train_index, "TEST:", test_index)
    ...     X_train, X_test = X[train_index], X[test_index]
    ...     y_train, y_test = y[train_index], y[test_index]
    TRAIN: [1 3] TEST: [0 2]
    TRAIN: [0 2] TEST: [1 3]

    Notes
    -----
    The implementation is designed to:

    * Generate test sets such that all contain the same distribution of
      classes, or as close as possible.

```

```

* Be invariant to class label: relabelling ``y = ["Happy", "Sad"]`` to
* ``y = [1, 0]`` should not change the indices generated.
* Preserve order dependencies in the dataset ordering, when
  ``shuffle=False``: all samples from class k in some test set were
  contiguous in y, or separated in y by samples from classes other than k.
* Generate test sets where the smallest and largest differ by at most one
  sample.

.. versionchanged:: 0.22
    The previous implementation did not follow the last constraint.

See also
-----
RepeatedStratifiedKFold: Repeats Stratified K-Fold n times.
"""

 @_deprecate_positional_args
def __init__(self, n_splits=5, *, shuffle=False, random_state=None):
    super().__init__(n_splits=n_splits, shuffle=shuffle,
                     random_state=random_state)

def _make_test_folds(self, X, y=None):
    rng = check_random_state(self.random_state)
    y = np.asarray(y)
    type_of_target_y = type_of_target(y)
    allowed_target_types = ('binary', 'multiclass')
    if type_of_target_y not in allowed_target_types:
        raise ValueError(
            'Supported target types are: {}. Got {!r} instead.'.format(
                allowed_target_types, type_of_target_y))

    y = column_or_1d(y)

    _, y_idx, y_inv = np.unique(y, return_index=True, return_inverse=True)
    # y_inv encodes y according to lexicographic order. We invert y_idx to
    # map the classes so that they are encoded by order of appearance:
    # 0 represents the first label appearing in y, 1 the second, etc.
    _, class_perm = np.unique(y_idx, return_inverse=True)
    y_encoded = class_perm[y_inv]

    n_classes = len(y_idx)
    y_counts = np.bincount(y_encoded)
    min_groups = np.min(y_counts)
    if np.all(self.n_splits > y_counts):
        raise ValueError("n_splits=%d cannot be greater than the"
                         " number of members in each class."
                         "% (self.n_splits))")
    if self.n_splits > min_groups:
        warnings.warn(("The least populated class in y has only %d"
                      " members, which is less than n_splits=%d."
                      "% (min_groups, self.n_splits)), UserWarning)

    # Determine the optimal number of samples from each class in each fold,
    # using round robin over the sorted y. (This can be done direct from
    # counts, but that code is unreadable.)
    y_order = np.sort(y_encoded)
    allocation = np.asarray([
        np.bincount(y_order[i::self.n_splits], minlength=n_classes)
        for i in range(self.n_splits)])
    # To maintain the data order dependencies as best as possible within
    # the stratification constraint, we assign samples from each class in
    # blocks (and then mess that up when shuffle=True).
    test_folds = np.empty(len(y), dtype='i')

```

```

for k in range(n_classes):
    # since the kth column of allocation stores the number of samples
    # of class k in each test set, this generates blocks of fold
    # indices corresponding to the allocation for class k.
    folds_for_class = np.arange(self.n_splits).repeat(allocation[:, k])
    if self.shuffle:
        rng.shuffle(folds_for_class)
    test_folds[y_encoded == k] = folds_for_class
return test_folds

def _iter_test_masks(self, X, y=None, groups=None):
    test_folds = self._make_test_folds(X, y)
    for i in range(self.n_splits):
        yield test_folds == i

def split(self, X, y, groups=None):
    """Generate indices to split data into training and test set.

    Parameters
    -----
    X : array-like of shape (n_samples, n_features)
        Training data, where n_samples is the number of samples
        and n_features is the number of features.

        Note that providing ``y`` is sufficient to generate the splits and
        hence ``np.zeros(n_samples)`` may be used as a placeholder for
        ``X`` instead of actual training data.

    y : array-like of shape (n_samples,)
        The target variable for supervised learning problems.
        Stratification is done based on the y labels.

    groups : object
        Always ignored, exists for compatibility.

    Yields
    -----
    train : ndarray
        The training set indices for that split.

    test : ndarray
        The testing set indices for that split.

    Notes
    -----
    Randomized CV splitters may return different results for each call of
    split. You can make the results identical by setting `random_state`
    to an integer.

    """
    y = check_array(y, ensure_2d=False, dtype=None)
    return super().split(X, y, groups)

class TimeSeriesSplit(_BaseKFold):
    """Time Series cross-validator

    Provides train/test indices to split time series data samples
    that are observed at fixed time intervals, in train/test sets.
    In each split, test indices must be higher than before, and thus shuffling
    in cross validator is inappropriate.

    This cross-validation object is a variation of :class:`KFold`.
    In the kth split, it returns first k folds as train set and the

```

```
(k+1)th fold as test set.
```

Note that unlike standard cross-validation methods, successive training sets are supersets of those that come before them.

Read more in the :ref:`User Guide <cross_validation>`.

Parameters

n_splits : int, default=5

Number of splits. Must be at least 2.

.. versionchanged:: 0.22

``n_splits`` default value changed from 3 to 5.

max_train_size : int, default=None

Maximum size for a single training set.

Examples

```
>>> import numpy as np
>>> from sklearn.model_selection import TimeSeriesSplit
>>> X = np.array([[1, 2], [3, 4], [1, 2], [3, 4], [1, 2], [3, 4]])
>>> y = np.array([1, 2, 3, 4, 5, 6])
>>> tscv = TimeSeriesSplit()
>>> print(tscv)
TimeSeriesSplit(max_train_size=None, n_splits=5)
>>> for train_index, test_index in tscv.split(X):
...     print("TRAIN:", train_index, "TEST:", test_index)
...     X_train, X_test = X[train_index], X[test_index]
...     y_train, y_test = y[train_index], y[test_index]
TRAIN: [0] TEST: [1]
TRAIN: [0 1] TEST: [2]
TRAIN: [0 1 2] TEST: [3]
TRAIN: [0 1 2 3] TEST: [4]
TRAIN: [0 1 2 3 4] TEST: [5]
```

Notes

The training set has size ``i * n_samples // (n_splits + 1) + n_samples % (n_splits + 1)`` in the ``i``th split, with a test set of size ``n_samples//(n_splits + 1)``, where ``n_samples`` is the number of samples.

"""

@_deprecate_positional_args

```
def __init__(self, n_splits=5, *, max_train_size=None):
    super().__init__(n_splits, shuffle=False, random_state=None)
    self.max_train_size = max_train_size
```

```
def split(self, X, y=None, groups=None):
```

"""Generate indices to split data into training and test set.

Parameters

X : array-like of shape (n_samples, n_features)

Training data, where n_samples is the number of samples and n_features is the number of features.

y : array-like of shape (n_samples,)

Always ignored, exists for compatibility.

groups : array-like of shape (n_samples,)

Always ignored, exists for compatibility.

```

Yields
-----
train : ndarray
    The training set indices for that split.

test : ndarray
    The testing set indices for that split.
"""
X, y, groups = indexable(X, y, groups)
n_samples = _num_samples(X)
n_splits = self.n_splits
n_folds = n_splits + 1
if n_folds > n_samples:
    raise ValueError(
        ("Cannot have number of folds ={0} greater"
         " than the number of samples: {1}.") .format(n_folds,
                                                       n_samples))
indices = np.arange(n_samples)
test_size = (n_samples // n_folds)
test_starts = range(test_size + n_samples % n_folds,
                     n_samples, test_size)
for test_start in test_starts:
    if self.max_train_size and self.max_train_size < test_start:
        yield (indices[test_start - self.max_train_size:test_start],
               indices[test_start:test_start + test_size])
    else:
        yield (indices[:test_start],
               indices[test_start:test_start + test_size])

```

```

class LeaveOneGroupOut(BaseCrossValidator):
    """Leave One Group Out cross-validator

```

Provides train/test indices to split data according to a third-party provided group. This group information can be used to encode arbitrary domain specific stratifications of the samples as integers.

For instance the groups could be the year of collection of the samples and thus allow for cross-validation against time-based splits.

Read more in the :ref:`User Guide <cross_validation>`.

Examples

```

>>> import numpy as np
>>> from sklearn.model_selection import LeaveOneGroupOut
>>> X = np.array([[1, 2], [3, 4], [5, 6], [7, 8]])
>>> y = np.array([1, 2, 1, 2])
>>> groups = np.array([1, 1, 2, 2])
>>> logo = LeaveOneGroupOut()
>>> logo.get_n_splits(X, y, groups)
2
>>> logo.get_n_splits(groups=groups) # 'groups' is always required
2
>>> print(logo)
LeaveOneGroupOut()
>>> for train_index, test_index in logo.split(X, y, groups):
...     print("TRAIN:", train_index, "TEST:", test_index)
...     X_train, X_test = X[train_index], X[test_index]
...     y_train, y_test = y[train_index], y[test_index]
...     print(X_train, X_test, y_train, y_test)
TRAIN: [2 3] TEST: [0 1]
```

```

[[5 6]
 [7 8]] [[1 2]
 [3 4]] [1 2] [1 2]
TRAIN: [0 1] TEST: [2 3]
[[1 2]
 [3 4]] [[5 6]
 [7 8]] [1 2] [1 2]

"""

def _iter_test_masks(self, X, y, groups):
    if groups is None:
        raise ValueError("The 'groups' parameter should not be None.")
    # We make a copy of groups to avoid side-effects during iteration
    groups = check_array(groups, copy=True, ensure_2d=False, dtype=None)
    unique_groups = np.unique(groups)
    if len(unique_groups) <= 1:
        raise ValueError(
            "The groups parameter contains fewer than 2 unique groups "
            "(%s). LeaveOneGroupOut expects at least 2." % unique_groups)
    for i in unique_groups:
        yield groups == i

def get_n_splits(self, X=None, y=None, groups=None):
    """Returns the number of splitting iterations in the cross-validator

    Parameters
    -----
    X : object
        Always ignored, exists for compatibility.

    y : object
        Always ignored, exists for compatibility.

    groups : array-like of shape (n_samples,)
        Group labels for the samples used while splitting the dataset into
        train/test set. This 'groups' parameter must always be specified to
        calculate the number of splits, though the other parameters can be
        omitted.

    Returns
    -----
    n_splits : int
        Returns the number of splitting iterations in the cross-validator.
    """
    if groups is None:
        raise ValueError("The 'groups' parameter should not be None.")
    groups = check_array(groups, ensure_2d=False, dtype=None)
    return len(np.unique(groups))

def split(self, X, y=None, groups=None):
    """Generate indices to split data into training and test set.

    Parameters
    -----
    X : array-like of shape (n_samples, n_features)
        Training data, where n_samples is the number of samples
        and n_features is the number of features.

    y : array-like of shape (n_samples,), default=None
        The target variable for supervised learning problems.

    groups : array-like of shape (n_samples,)

```

```

    Group labels for the samples used while splitting the dataset into
    train/test set.

Yields
-----
train : ndarray
    The training set indices for that split.

test : ndarray
    The testing set indices for that split.
"""
return super().split(X, y, groups)

class LeavePGroupsOut(BaseCrossValidator):
    """Leave P Group(s) Out cross-validator

Provides train/test indices to split data according to a third-party
provided group. This group information can be used to encode arbitrary
domain specific stratifications of the samples as integers.

For instance the groups could be the year of collection of the samples
and thus allow for cross-validation against time-based splits.

The difference between LeavePGroupsOut and LeaveOneGroupOut is that
the former builds the test sets with all the samples assigned to
``p`` different values of the groups while the latter uses samples
all assigned the same groups.

Read more in the :ref:`User Guide <cross_validation>`.

Parameters
-----
n_groups : int
    Number of groups (``p``) to leave out in the test split.

Examples
-----
>>> import numpy as np
>>> from sklearn.model_selection import LeavePGroupsOut
>>> X = np.array([[1, 2], [3, 4], [5, 6]])
>>> y = np.array([1, 2, 1])
>>> groups = np.array([1, 2, 3])
>>> lrgo = LeavePGroupsOut(n_groups=2)
>>> lrgo.get_n_splits(X, y, groups)
3
>>> lrgo.get_n_splits(groups=groups) # 'groups' is always required
3
>>> print(lrgo)
LeavePGroupsOut(n_groups=2)
>>> for train_index, test_index in lrgo.split(X, y, groups):
...     print("TRAIN:", train_index, "TEST:", test_index)
...     X_train, X_test = X[train_index], X[test_index]
...     y_train, y_test = y[train_index], y[test_index]
...     print(X_train, X_test, y_train, y_test)
TRAIN: [2] TEST: [0 1]
[[5 6]] [[1 2]
 [3 4]] [1] [1 2]
TRAIN: [1] TEST: [0 2]
[[3 4]] [[1 2]
 [5 6]] [2] [1 1]
TRAIN: [0] TEST: [1 2]
[[1 2]] [[3 4]]

```

```
[5 6] [1] [2 1]
```

See also

GroupKFold: K-fold iterator variant with non-overlapping groups.

"""

```
def __init__(self, n_groups):
    self.n_groups = n_groups

def _iter_test_masks(self, X, y, groups):
    if groups is None:
        raise ValueError("The 'groups' parameter should not be None.")
    groups = check_array(groups, copy=True, ensure_2d=False, dtype=None)
    unique_groups = np.unique(groups)
    if self.n_groups >= len(unique_groups):
        raise ValueError(
            "The groups parameter contains fewer than (or equal to) "
            "%d numbers of unique groups (%s). LeavePGroupsOut "
            "expects that at least %d unique groups be "
            "present" % (self.n_groups, unique_groups, self.n_groups + 1))
    combi = combinations(range(len(unique_groups)), self.n_groups)
    for indices in combi:
        test_index = np.zeros(_num_samples(X), dtype=np.bool)
        for l in unique_groups[np.array(indices)]:
            test_index[groups == l] = True
        yield test_index
```

```
def get_n_splits(self, X=None, y=None, groups=None):
```

"""Returns the number of splitting iterations in the cross-validator

Parameters

X : object

Always ignored, exists for compatibility.

y : object

Always ignored, exists for compatibility.

groups : array-like of shape (n_samples,)

Group labels for the samples used while splitting the dataset into train/test set. This 'groups' parameter must always be specified to calculate the number of splits, though the other parameters can be omitted.

Returns

n_splits : int

Returns the number of splitting iterations in the cross-validator.

"""

if groups is None:

raise ValueError("The 'groups' parameter should not be None.")

groups = check_array(groups, ensure_2d=False, dtype=None)

return int(comb(len(np.unique(groups)), self.n_groups, exact=True))

```
def split(self, X, y=None, groups=None):
```

"""Generate indices to split data into training and test set.

Parameters

X : array-like of shape (n_samples, n_features)

Training data, where n_samples is the number of samples and n_features is the number of features.

```

y : array-like of shape (n_samples,), default=None
    The target variable for supervised learning problems.

groups : array-like of shape (n_samples,)
    Group labels for the samples used while splitting the dataset into
    train/test set.

Yields
-----
train : ndarray
    The training set indices for that split.

test : ndarray
    The testing set indices for that split.
"""
return super().split(X, y, groups)

class _RepeatedSplits(metaclass=ABCMeta):
    """Repeated splits for an arbitrary randomized CV splitter.

    Repeats splits for cross-validation n times with different randomization
    in each repetition.

    Parameters
    -----
    cv : callable
        Cross-validator class.

    n_repeats : int, default=10
        Number of times cross-validator needs to be repeated.

    random_state : int or RandomState instance, default=None
        Passes `random_state` to the arbitrary repeating cross validator.
        Pass an int for reproducible output across multiple function calls.
        See :term:`Glossary <random_state>`.

    **cvargs : additional params
        Constructor parameters for cv. Must not contain random_state
        and shuffle.

    """
    @_deprecated_positional_args
    def __init__(self, cv, *, n_repeats=10, random_state=None, **cvargs):
        if not isinstance(n_repeats, numbers.Integral):
            raise ValueError("Number of repetitions must be of Integral type.")

        if n_repeats <= 0:
            raise ValueError("Number of repetitions must be greater than 0.")

        if any(key in cvargs for key in ('random_state', 'shuffle')):
            raise ValueError(
                "cvargs must not contain random_state or shuffle.")

        self.cv = cv
        self.n_repeats = n_repeats
        self.random_state = random_state
        self.cvargs = cvargs

    def split(self, X, y=None, groups=None):
        """Generates indices to split data into training and test set.

        Parameters
        """

```

```

-----
X : array-like, shape (n_samples, n_features)
    Training data, where n_samples is the number of samples
    and n_features is the number of features.

y : array-like of length n_samples
    The target variable for supervised learning problems.

groups : array-like of shape (n_samples,), default=None
    Group labels for the samples used while splitting the dataset into
    train/test set.

Yields
-----
train : ndarray
    The training set indices for that split.

test : ndarray
    The testing set indices for that split.
"""
n_repeats = self.n_repeats
rng = check_random_state(self.random_state)

for idx in range(n_repeats):
    cv = self.cv(random_state=rng, shuffle=True,
                 **self.cvargs)
    for train_index, test_index in cv.split(X, y, groups):
        yield train_index, test_index

def get_n_splits(self, X=None, y=None, groups=None):
    """Returns the number of splitting iterations in the cross-validator

Parameters
-----
X : object
    Always ignored, exists for compatibility.
    ``np.zeros(n_samples)`` may be used as a placeholder.

y : object
    Always ignored, exists for compatibility.
    ``np.zeros(n_samples)`` may be used as a placeholder.

groups : array-like of shape (n_samples,), default=None
    Group labels for the samples used while splitting the dataset into
    train/test set.

Returns
-----
n_splits : int
    Returns the number of splitting iterations in the cross-validator.
"""
    rng = check_random_state(self.random_state)
    cv = self.cv(random_state=rng, shuffle=True,
                 **self.cvargs)
    return cv.get_n_splits(X, y, groups) * self.n_repeats

def __repr__(self):
    return _build_repr(self)

class RepeatedKFold(_RepeatedSplits):
    """Repeated K-Fold cross validator.

```

```
Repeats K-Fold n times with different randomization in each repetition.
```

Read more in the :ref:`User Guide <cross_validation>`.

Parameters

```
n_splits : int, default=5  
    Number of folds. Must be at least 2.
```

```
n_repeats : int, default=10  
    Number of times cross-validator needs to be repeated.
```

```
random_state : int or RandomState instance, default=None  
    Controls the randomness of each repeated cross-validation instance.  
    Pass an int for reproducible output across multiple function calls.  
    See :term:`Glossary <random_state>`.
```

Examples

```
>>> import numpy as np  
>>> from sklearn.model_selection import RepeatedKFold  
>>> X = np.array([[1, 2], [3, 4], [1, 2], [3, 4]])  
>>> y = np.array([0, 0, 1, 1])  
>>> rkf = RepeatedKFold(n_splits=2, n_repeats=2, random_state=2652124)  
>>> for train_index, test_index in rkf.split(X):  
...     print("TRAIN:", train_index, "TEST:", test_index)  
...     X_train, X_test = X[train_index], X[test_index]  
...     y_train, y_test = y[train_index], y[test_index]  
...  
TRAIN: [0 1] TEST: [2 3]  
TRAIN: [2 3] TEST: [0 1]  
TRAIN: [1 2] TEST: [0 3]  
TRAIN: [0 3] TEST: [1 2]
```

Notes

Randomized CV splitters may return different results for each call of `split`. You can make the results identical by setting `'random_state'` to an integer.

See also

```
RepeatedStratifiedKFold: Repeats Stratified K-Fold n times.
```

```
"""
```

```
@_deprecate_positional_args  
def __init__(self, *, n_splits=5, n_repeats=10, random_state=None):  
    super().__init__()  
    KFold, n_repeats=n_repeats,  
    random_state=random_state, n_splits=n_splits)
```

```
class RepeatedStratifiedKFold(_RepeatedSplits):  
    """Repeated Stratified K-Fold cross validator.
```

Repeats Stratified K-Fold n times with different randomization in each repetition.

Read more in the :ref:`User Guide <cross_validation>`.

Parameters

```
n_splits : int, default=5  
    Number of folds. Must be at least 2.
```

```
n_repeats : int, default=10
    Number of times cross-validator needs to be repeated.

random_state : int or RandomState instance, default=None
    Controls the generation of the random states for each repetition.
    Pass an int for reproducible output across multiple function calls.
    See :term:`Glossary <random_state>`.
```

Examples

```
>>> import numpy as np
>>> from sklearn.model_selection import RepeatedStratifiedKFold
>>> X = np.array([[1, 2], [3, 4], [1, 2], [3, 4]])
>>> y = np.array([0, 0, 1, 1])
>>> rskf = RepeatedStratifiedKFold(n_splits=2, n_repeats=2,
...     random_state=36851234)
>>> for train_index, test_index in rskf.split(X, y):
...     print("TRAIN:", train_index, "TEST:", test_index)
...     X_train, X_test = X[train_index], X[test_index]
...     y_train, y_test = y[train_index], y[test_index]
...
TRAIN: [1 2] TEST: [0 3]
TRAIN: [0 3] TEST: [1 2]
TRAIN: [1 3] TEST: [0 2]
TRAIN: [0 2] TEST: [1 3]
```

Notes

Randomized CV splitters may return different results for each call of `split`. You can make the results identical by setting `'random_state'` to an integer.

See also

```
RepeatedKFold: Repeats K-Fold n times.
```

```
"""
@_deprecate_positional_args
def __init__(self, *, n_splits=5, n_repeats=10, random_state=None):
    super().__init__(
        StratifiedKFold, n_repeats=n_repeats, random_state=random_state,
        n_splits=n_splits)

class BaseShuffleSplit(metaclass=ABCMeta):
    """Base class for ShuffleSplit and StratifiedShuffleSplit"""
    @_deprecate_positional_args
    def __init__(self, n_splits=10, *, test_size=None, train_size=None,
                 random_state=None):
        self.n_splits = n_splits
        self.test_size = test_size
        self.train_size = train_size
        self.random_state = random_state
        self._default_test_size = 0.1

    def split(self, X, y=None, groups=None):
        """Generate indices to split data into training and test set.

Parameters
-----
X : array-like of shape (n_samples, n_features)
    Training data, where n_samples is the number of samples
    and n_features is the number of features.
```

```
y : array-like of shape (n_samples,)
    The target variable for supervised learning problems.

groups : array-like of shape (n_samples,), default=None
    Group labels for the samples used while splitting the dataset into
    train/test set.

Yields
-----
train : ndarray
    The training set indices for that split.

test : ndarray
    The testing set indices for that split.

Notes
-----
Randomized CV splitters may return different results for each call of
split. You can make the results identical by setting `random_state`
to an integer.

"""
X, y, groups = indexable(X, y, groups)
for train, test in self._iter_indices(X, y, groups):
    yield train, test

@abstractmethod
def _iter_indices(self, X, y=None, groups=None):
    """Generate (train, test) indices"""

def get_n_splits(self, X=None, y=None, groups=None):
    """Returns the number of splitting iterations in the cross-validator

Parameters
-----
X : object
    Always ignored, exists for compatibility.

y : object
    Always ignored, exists for compatibility.

groups : object
    Always ignored, exists for compatibility.

Returns
-----
n_splits : int
    Returns the number of splitting iterations in the cross-validator.

"""
    return self.n_splits

def __repr__(self):
    return _build_repr(self)

class ShuffleSplit(BaseShuffleSplit):
    """Random permutation cross-validator

Yields indices to split data into training and test sets.

Note: contrary to other cross-validation strategies, random splits
do not guarantee that all folds will be different, although this is
still very likely for sizeable datasets.
```

```
Read more in the :ref:`User Guide <cross_validation>`.
```

Parameters

```
n_splits : int, default=10
    Number of re-shuffling & splitting iterations.

test_size : float or int, default=None
    If float, should be between 0.0 and 1.0 and represent the proportion
    of the dataset to include in the test split. If int, represents the
    absolute number of test samples. If None, the value is set to the
    complement of the train size. If ``train_size`` is also None, it will
    be set to 0.1.

train_size : float or int, default=None
    If float, should be between 0.0 and 1.0 and represent the
    proportion of the dataset to include in the train split. If
    int, represents the absolute number of train samples. If None,
    the value is automatically set to the complement of the test size.

random_state : int or RandomState instance, default=None
    Controls the randomness of the training and testing indices produced.
    Pass an int for reproducible output across multiple function calls.
    See :term:`Glossary <random_state>`.
```

Examples

```
>>> import numpy as np
>>> from sklearn.model_selection import ShuffleSplit
>>> X = np.array([[1, 2], [3, 4], [5, 6], [7, 8], [3, 4], [5, 6]])
>>> y = np.array([1, 2, 1, 2, 1, 2])
>>> rs = ShuffleSplit(n_splits=5, test_size=.25, random_state=0)
>>> rs.get_n_splits(X)
5
>>> print(rs)
ShuffleSplit(n_splits=5, random_state=0, test_size=0.25, train_size=None)
>>> for train_index, test_index in rs.split(X):
...     print("TRAIN:", train_index, "TEST:", test_index)
TRAIN: [1 3 0 4] TEST: [5 2]
TRAIN: [4 0 2 5] TEST: [1 3]
TRAIN: [1 2 4 0] TEST: [3 5]
TRAIN: [3 4 1 0] TEST: [5 2]
TRAIN: [3 5 1 0] TEST: [2 4]
>>> rs = ShuffleSplit(n_splits=5, train_size=0.5, test_size=.25,
...                     random_state=0)
>>> for train_index, test_index in rs.split(X):
...     print("TRAIN:", train_index, "TEST:", test_index)
TRAIN: [1 3 0] TEST: [5 2]
TRAIN: [4 0 2] TEST: [1 3]
TRAIN: [1 2 4] TEST: [3 5]
TRAIN: [3 4 1] TEST: [5 2]
TRAIN: [3 5 1] TEST: [2 4]
"""
 @_deprecate_positional_args
def __init__(self, n_splits=10, *, test_size=None, train_size=None,
             random_state=None):
    super().__init__(
        n_splits=n_splits,
        test_size=test_size,
        train_size=train_size,
        random_state=random_state)
    self._default_test_size = 0.1
```

```

def _iter_indices(self, X, y=None, groups=None):
    n_samples = _num_samples(X)
    n_train, n_test = _validate_shuffle_split(
        n_samples, self.test_size, self.train_size,
        default_test_size=self._default_test_size)

    rng = check_random_state(self.random_state)
    for i in range(self.n_splits):
        # random partition
        permutation = rng.permutation(n_samples)
        ind_test = permutation[:n_test]
        ind_train = permutation[n_test:(n_test + n_train)]
        yield ind_train, ind_test


class GroupShuffleSplit(ShuffleSplit):
    '''Shuffle-Group(s)-Out cross-validation iterator
    Provides randomized train/test indices to split data according to a
    third-party provided group. This group information can be used to encode
    arbitrary domain specific stratifications of the samples as integers.

    For instance the groups could be the year of collection of the samples
    and thus allow for cross-validation against time-based splits.

    The difference between LeavePGroupsOut and GroupShuffleSplit is that
    the former generates splits using all subsets of size ``p`` unique groups,
    whereas GroupShuffleSplit generates a user-determined number of random
    test splits, each with a user-determined fraction of unique groups.

    For example, a less computationally intensive alternative to
    ``LeavePGroupsOut(p=10)`` would be
    ``GroupShuffleSplit(test_size=10, n_splits=100)``.

    Note: The parameters ``test_size`` and ``train_size`` refer to groups, and
    not to samples, as in ShuffleSplit.

```

Parameters

n_splits : int, default=5
 Number of re-shuffling & splitting iterations.

test_size : float, int, default=0.2
 If float, should be between 0.0 and 1.0 and represent the proportion
 of groups to include in the test split (rounded up). If int,
 represents the absolute number of test groups. If None, the value is
 set to the complement of the train size.
 The default will change in version 0.21. It will remain 0.2 only
 if ``train_size`` is unspecified, otherwise it will complement
 the specified ``train_size``.

train_size : float or int, default=None
 If float, should be between 0.0 and 1.0 and represent the
 proportion of the groups to include in the train split. If
 int, represents the absolute number of train groups. If None,
 the value is automatically set to the complement of the test size.

random_state : int or RandomState instance, default=None
 Controls the randomness of the training and testing indices produced.
 Pass an int for reproducible output across multiple function calls.
 See :term:`Glossary <random_state>`.

Examples

```
>>> import numpy as np
>>> from sklearn.model_selection import GroupShuffleSplit
>>> X = np.ones(shape=(8, 2))
>>> y = np.ones(shape=(8, 1))
>>> groups = np.array([1, 1, 2, 2, 2, 3, 3, 3])
>>> print(groups.shape)
(8,)
>>> gss = GroupShuffleSplit(n_splits=2, train_size=.7, random_state=42)
>>> gss.get_n_splits()
2
>>> for train_idx, test_idx in gss.split(X, y, groups):
...     print("TRAIN:", train_idx, "TEST:", test_idx)
TRAIN: [2 3 4 5 6 7] TEST: [0 1]
TRAIN: [0 1 5 6 7] TEST: [2 3 4]
...
@_deprecate_positional_args
def __init__(self, n_splits=5, *, test_size=None, train_size=None,
            random_state=None):
    super().__init__(
        n_splits=n_splits,
        test_size=test_size,
        train_size=train_size,
        random_state=random_state)
    self._default_test_size = 0.2

def __iter_indices(self, X, y, groups):
    if groups is None:
        raise ValueError("The 'groups' parameter should not be None.")
    groups = check_array(groups, ensure_2d=False, dtype=None)
    classes, group_indices = np.unique(groups, return_inverse=True)
    for group_train, group_test in super().__iter_indices(X=classes):
        # these are the indices of classes in the partition
        # invert them into data indices

        train = np.flatnonzero(np.in1d(group_indices, group_train))
        test = np.flatnonzero(np.in1d(group_indices, group_test))

        yield train, test

def split(self, X, y=None, groups=None):
    """Generate indices to split data into training and test set.

Parameters


-----



X : array-like of shape (n_samples, n_features)  

   Training data, where n_samples is the number of samples  

   and n_features is the number of features.



y : array-like of shape (n_samples,), default=None  

   The target variable for supervised learning problems.



groups : array-like of shape (n_samples,)  

   Group labels for the samples used while splitting the dataset into  

   train/test set.



Yields



-----



train : ndarray  

   The training set indices for that split.


```

```

    test : ndarray
        The testing set indices for that split.

    Notes
    -----
    Randomized CV splitters may return different results for each call of
    split. You can make the results identical by setting `random_state`
    to an integer.

    """
    return super().split(X, y, groups)

class StratifiedShuffleSplit(BaseShuffleSplit):
    """Stratified ShuffleSplit cross-validator

    Provides train/test indices to split data in train/test sets.

    This cross-validation object is a merge of StratifiedKFold and
    ShuffleSplit, which returns stratified randomized folds. The folds
    are made by preserving the percentage of samples for each class.

    Note: like the ShuffleSplit strategy, stratified random splits
    do not guarantee that all folds will be different, although this is
    still very likely for sizeable datasets.

    Read more in the :ref:`User Guide <cross_validation>`.

    Parameters
    -----
    n_splits : int, default=10
        Number of re-shuffling & splitting iterations.

    test_size : float or int, default=None
        If float, should be between 0.0 and 1.0 and represent the proportion
        of the dataset to include in the test split. If int, represents the
        absolute number of test samples. If None, the value is set to the
        complement of the train size. If ``train_size`` is also None, it will
        be set to 0.1.

    train_size : float or int, default=None
        If float, should be between 0.0 and 1.0 and represent the
        proportion of the dataset to include in the train split. If
        int, represents the absolute number of train samples. If None,
        the value is automatically set to the complement of the test size.

    random_state : int or RandomState instance, default=None
        Controls the randomness of the training and testing indices produced.
        Pass an int for reproducible output across multiple function calls.
        See :term:`Glossary <random_state>`.

    Examples
    -----
    >>> import numpy as np
    >>> from sklearn.model_selection import StratifiedShuffleSplit
    >>> X = np.array([[1, 2], [3, 4], [1, 2], [3, 4], [1, 2], [3, 4]])
    >>> y = np.array([0, 0, 0, 1, 1, 1])
    >>> sss = StratifiedShuffleSplit(n_splits=5, test_size=0.5, random_state=0)
    >>> sss.get_n_splits(X, y)
    5
    >>> print(sss)
    StratifiedShuffleSplit(n_splits=5, random_state=0, ...)
    >>> for train_index, test_index in sss.split(X, y):
    ...     print("TRAIN:", train_index, "TEST:", test_index)

```

```

...      X_train, X_test = X[train_index], X[test_index]
...      y_train, y_test = y[train_index], y[test_index]
TRAIN: [5 2 3] TEST: [4 1 0]
TRAIN: [5 1 4] TEST: [0 2 3]
TRAIN: [5 0 2] TEST: [4 3 1]
TRAIN: [4 1 0] TEST: [2 3 5]
TRAIN: [0 5 1] TEST: [3 4 2]
"""
 @_deprecate_positional_args
def __init__(self, n_splits=10, *, test_size=None, train_size=None,
             random_state=None):
    super().__init__(
        n_splits=n_splits,
        test_size=test_size,
        train_size=train_size,
        random_state=random_state)
    self._default_test_size = 0.1

def _iter_indices(self, X, y, groups=None):
    n_samples = _num_samples(X)
    y = check_array(y, ensure_2d=False, dtype=None)
    n_train, n_test = _validate_shuffle_split(
        n_samples, self.test_size, self.train_size,
        default_test_size=self._default_test_size)

    if y.ndim == 2:
        # for multi-label y, map each distinct row to a string repr
        # using join because str(row) uses an ellipsis if len(row) > 1000
        y = np.array([' '.join(row.astype('str')) for row in y])

    classes, y_indices = np.unique(y, return_inverse=True)
    n_classes = classes.shape[0]

    class_counts = np.bincount(y_indices)
    if np.min(class_counts) < 2:
        raise ValueError("The least populated class in y has only 1"
                         " member, which is too few. The minimum"
                         " number of groups for any class cannot"
                         " be less than 2.")

    if n_train < n_classes:
        raise ValueError('The train_size = %d should be greater or '
                         'equal to the number of classes = %d' %
                         (n_train, n_classes))
    if n_test < n_classes:
        raise ValueError('The test_size = %d should be greater or '
                         'equal to the number of classes = %d' %
                         (n_test, n_classes))

    # Find the sorted list of instances for each class:
    # (np.unique above performs a sort, so code is O(n logn) already)
    class_indices = np.split(np.argsort(y_indices, kind='mergesort'),
                            np.cumsum(class_counts)[:-1])

    rng = check_random_state(self.random_state)

    for _ in range(self.n_splits):
        # if there are ties in the class-counts, we want
        # to make sure to break them anew in each iteration
        n_i = _approximate_mode(class_counts, n_train, rng)
        class_counts_remaining = class_counts - n_i
        t_i = _approximate_mode(class_counts_remaining, n_test, rng)

```

```

train = []
test = []

    for i in range(n_classes):
        permutation = rng.permutation(class_counts[i])
        perm_indices_class_i = class_indices[i].take(permutation,
                                                      mode='clip')

        train.extend(perm_indices_class_i[:n_i[i]])
        test.extend(perm_indices_class_i[n_i[i]:n_i[i] + t_i[i]])

    train = rng.permutation(train)
    test = rng.permutation(test)

    yield train, test

def split(self, X, y, groups=None):
    """Generate indices to split data into training and test set.

    Parameters
    -----
    X : array-like of shape (n_samples, n_features)
        Training data, where n_samples is the number of samples
        and n_features is the number of features.

        Note that providing ``y`` is sufficient to generate the splits and
        hence ``np.zeros(n_samples)`` may be used as a placeholder for
        ``X`` instead of actual training data.

    y : array-like of shape (n_samples,) or (n_samples, n_labels)
        The target variable for supervised learning problems.
        Stratification is done based on the y labels.

    groups : object
        Always ignored, exists for compatibility.

    Yields
    -----
    train : ndarray
        The training set indices for that split.

    test : ndarray
        The testing set indices for that split.

    Notes
    -----
    Randomized CV splitters may return different results for each call of
    split. You can make the results identical by setting `random_state`
    to an integer.

    """
    y = check_array(y, ensure_2d=False, dtype=None)
    return super().split(X, y, groups)

def _validate_shuffle_split(n_samples, test_size, train_size,
                           default_test_size=None):
    """
    Validation helper to check if the test/test sizes are meaningful wrt to the
    size of the data (n_samples)
    """
    if test_size is None and train_size is None:
        test_size = default_test_size

```

```

test_size_type = np.asarray(test_size).dtype.kind
train_size_type = np.asarray(train_size).dtype.kind

if (test_size_type == 'i' and (test_size >= n_samples or test_size <= 0)
    or test_size_type == 'f' and (test_size <= 0 or test_size >= 1)):
    raise ValueError('test_size={} should be either positive and smaller'
                     ' than the number of samples {} or a float in the '
                     '(0, 1) range'.format(test_size, n_samples))

if (train_size_type == 'i' and (train_size >= n_samples or train_size <= 0)
    or train_size_type == 'f' and (train_size <= 0 or train_size >= 1)):
    raise ValueError('train_size={} should be either positive and smaller'
                     ' than the number of samples {} or a float in the '
                     '(0, 1) range'.format(train_size, n_samples))

if train_size is not None and train_size_type not in ('i', 'f'):
    raise ValueError("Invalid value for train_size: {}".format(train_size))
if test_size is not None and test_size_type not in ('i', 'f'):
    raise ValueError("Invalid value for test_size: {}".format(test_size))

if (train_size_type == 'f' and test_size_type == 'f' and
    train_size + test_size > 1):
    raise ValueError(
        'The sum of test_size and train_size = {}, should be in the (0, 1)'
        ' range. Reduce test_size and/or train_size.'
        .format(train_size + test_size))

if test_size_type == 'f':
    n_test = ceil(test_size * n_samples)
elif test_size_type == 'i':
    n_test = float(test_size)

if train_size_type == 'f':
    n_train = floor(train_size * n_samples)
elif train_size_type == 'i':
    n_train = float(train_size)

if train_size is None:
    n_train = n_samples - n_test
elif test_size is None:
    n_test = n_samples - n_train

if n_train + n_test > n_samples:
    raise ValueError('The sum of train_size and test_size = %d, '
                     'should be smaller than the number of '
                     '%samples %d. Reduce test_size and/or '
                     'train_size.' % (n_train + n_test, n_samples))

n_train, n_test = int(n_train), int(n_test)

if n_train == 0:
    raise ValueError(
        'With n_samples={}, test_size={} and train_size={}, the '
        'resulting train set will be empty. Adjust any of the '
        'aforementioned parameters.'.format(n_samples, test_size,
                                             train_size)
    )

return n_train, n_test

class PredefinedSplit(BaseCrossValidator):
    """Predefined split cross-validator

```

Provides train/test indices to split data into train/test sets using a predefined scheme specified by the user with the ``test_fold`` parameter.

Read more in the :ref:`User Guide <cross_validation>`.

.. versionadded:: 0.16

Parameters

test_fold : array-like of shape (n_samples,)

The entry ``test_fold[i]`` represents the index of the test set that sample ``i`` belongs to. It is possible to exclude sample ``i`` from any test set (i.e. include sample ``i`` in every training set) by setting ``test_fold[i]`` equal to -1.

Examples

```
>>> import numpy as np
>>> from sklearn.model_selection import PredefinedSplit
>>> X = np.array([[1, 2], [3, 4], [1, 2], [3, 4]])
>>> y = np.array([0, 0, 1, 1])
>>> test_fold = [0, 1, -1, 1]
>>> ps = PredefinedSplit(test_fold)
>>> ps.get_n_splits()
2
>>> print(ps)
PredefinedSplit(test_fold=array([ 0,  1, -1,  1]))
>>> for train_index, test_index in ps.split():
...     print("TRAIN:", train_index, "TEST:", test_index)
...     X_train, X_test = X[train_index], X[test_index]
...     y_train, y_test = y[train_index], y[test_index]
TRAIN: [1 2 3] TEST: [0]
TRAIN: [0 2] TEST: [1 3]
"""
```

```
def __init__(self, test_fold):
```

```
    self.test_fold = np.array(test_fold, dtype=np.int)
    self.test_fold = column_or_1d(self.test_fold)
    self.unique_folds = np.unique(self.test_fold)
    self.unique_folds = self.unique_folds[self.unique_folds != -1]
```

```
def split(self, X=None, y=None, groups=None):
```

```
    """Generate indices to split data into training and test set.
```

Parameters

X : object

Always ignored, exists for compatibility.

y : object

Always ignored, exists for compatibility.

groups : object

Always ignored, exists for compatibility.

Yields

train : ndarray

The training set indices for that split.

test : ndarray

The testing set indices for that split.

```

"""
ind = np.arange(len(self.test_fold))
for test_index in self._iter_test_masks():
    train_index = ind[np.logical_not(test_index)]
    test_index = ind[test_index]
    yield train_index, test_index

def _iter_test_masks(self):
    """Generates boolean masks corresponding to test sets."""
    for f in self.unique_folds:
        test_index = np.where(self.test_fold == f)[0]
        test_mask = np.zeros(len(self.test_fold), dtype=np.bool)
        test_mask[test_index] = True
        yield test_mask

def get_n_splits(self, X=None, y=None, groups=None):
    """Returns the number of splitting iterations in the cross-validator

    Parameters
    -----
    X : object
        Always ignored, exists for compatibility.

    y : object
        Always ignored, exists for compatibility.

    groups : object
        Always ignored, exists for compatibility.

    Returns
    -----
    n_splits : int
        Returns the number of splitting iterations in the cross-validator.
    """
    return len(self.unique_folds)

class _CVIterableWrapper(BaseCrossValidator):
    """Wrapper class for old style cv objects and iterables."""
    def __init__(self, cv):
        self.cv = list(cv)

    def get_n_splits(self, X=None, y=None, groups=None):
        """Returns the number of splitting iterations in the cross-validator

        Parameters
        -----
        X : object
            Always ignored, exists for compatibility.

        y : object
            Always ignored, exists for compatibility.

        groups : object
            Always ignored, exists for compatibility.

        Returns
        -----
        n_splits : int
            Returns the number of splitting iterations in the cross-validator.
        """
        return len(self.cv)

```

```

def split(self, X=None, y=None, groups=None):
    """Generate indices to split data into training and test set.

    Parameters
    -----
    X : object
        Always ignored, exists for compatibility.

    y : object
        Always ignored, exists for compatibility.

    groups : object
        Always ignored, exists for compatibility.

    Yields
    -----
    train : ndarray
        The training set indices for that split.

    test : ndarray
        The testing set indices for that split.
    """
    for train, test in self.cv:
        yield train, test

@_deprecate_positional_args
def check_cv(cv=5, y=None, *, classifier=False):
    """Input checker utility for building a cross-validator

    Parameters
    -----
    cv : int, cross-validation generator or an iterable, default=None
        Determines the cross-validation splitting strategy.
        Possible inputs for cv are:
        - None, to use the default 5-fold cross validation,
        - integer, to specify the number of folds.
        - :term:`CV splitter`,
        - An iterable yielding (train, test) splits as arrays of indices.

        For integer/None inputs, if classifier is True and ``y`` is either
        binary or multiclass, :class:`StratifiedKFold` is used. In all other
        cases, :class:`KFold` is used.

        Refer :ref:`User Guide <cross_validation>` for the various
        cross-validation strategies that can be used here.

    .. versionchanged:: 0.22
        ``cv`` default value changed from 3-fold to 5-fold.

    y : array-like, default=None
        The target variable for supervised learning problems.

    classifier : bool, default=False
        Whether the task is a classification task, in which case
        stratified KFold will be used.

    Returns
    -----
    checked_cv : a cross-validator instance.
        The return value is a cross-validator which generates the train/test
        splits via the ``split`` method.
    """

```

```

cv = 5 if cv is None else cv
if isinstance(cv, numbers.Integral):
    if (classifier and (y is not None) and
        (type_of_target(y) in ('binary', 'multiclass'))):
        return StratifiedKFold(cv)
    else:
        return KFold(cv)

if not hasattr(cv, 'split') or isinstance(cv, str):
    if not isinstance(cv, Iterable) or isinstance(cv, str):
        raise ValueError("Expected cv as an integer, cross-validation "
                         "object (from sklearn.model_selection) "
                         "or an iterable. Got %s." % cv)
    return _CVIterableWrapper(cv)

return cv # New style cv objects are passed without any modification

def train_test_split(*arrays, **options):
    """Split arrays or matrices into random train and test subsets

    Quick utility that wraps input validation and
    ``next(ShuffleSplit().split(X, y))`` and application to input data
    into a single call for splitting (and optionally subsampling) data in a
    oneliner.

    Read more in the :ref:`User Guide <cross_validation>`.

    Parameters
    -----
    *arrays : sequence of indexables with same length / shape[0]
        Allowed inputs are lists, numpy arrays, scipy-sparse
        matrices or pandas dataframes.

    test_size : float or int, default=None
        If float, should be between 0.0 and 1.0 and represent the proportion
        of the dataset to include in the test split. If int, represents the
        absolute number of test samples. If None, the value is set to the
        complement of the train size. If ``train_size`` is also None, it will
        be set to 0.25.

    train_size : float or int, default=None
        If float, should be between 0.0 and 1.0 and represent the
        proportion of the dataset to include in the train split. If
        int, represents the absolute number of train samples. If None,
        the value is automatically set to the complement of the test size.

    random_state : int or RandomState instance, default=None
        Controls the shuffling applied to the data before applying the split.
        Pass an int for reproducible output across multiple function calls.
        See :term:`Glossary <random_state>`.

    shuffle : bool, default=True
        Whether or not to shuffle the data before splitting. If shuffle=False
        then stratify must be None.

    stratify : array-like, default=None
        If not None, data is split in a stratified fashion, using this as
        the class labels.

    Returns
    -----

```

```
splitting : list, length=2 * len(arrays)
    List containing train-test split of inputs.

    .. versionadded:: 0.16
        If the input is sparse, the output will be a
        ``scipy.sparse.csr_matrix``. Else, output type is the same as the
        input type.

Examples
-----
>>> import numpy as np
>>> from sklearn.model_selection import train_test_split
>>> X, y = np.arange(10).reshape((5, 2)), range(5)
>>> X
array([[0, 1],
       [2, 3],
       [4, 5],
       [6, 7],
       [8, 9]])
>>> list(y)
[0, 1, 2, 3, 4]

>>> X_train, X_test, y_train, y_test = train_test_split(
...     X, y, test_size=0.33, random_state=42)
...
>>> X_train
array([[4, 5],
       [0, 1],
       [6, 7]])
>>> y_train
[2, 0, 3]
>>> X_test
array([[2, 3],
       [8, 9]])
>>> y_test
[1, 4]

>>> train_test_split(y, shuffle=False)
[[0, 1, 2], [3, 4]]
```

"""

```
n_arrays = len(arrays)
if n_arrays == 0:
    raise ValueError("At least one array required as input")
test_size = options.pop('test_size', None)
train_size = options.pop('train_size', None)
random_state = options.pop('random_state', None)
stratify = options.pop('stratify', None)
shuffle = options.pop('shuffle', True)

if options:
    raise TypeError("Invalid parameters passed: %s" % str(options))

arrays = indexable(*arrays)

n_samples = _num_samples(arrays[0])
n_train, n_test = _validate_shuffle_split(n_samples, test_size, train_size,
                                         default_test_size=0.25)

if shuffle is False:
    if stratify is not None:
        raise ValueError(
            "Stratified train/test split is not implemented for "
```

```

        "shuffle=False")

train = np.arange(n_train)
test = np.arange(n_train, n_train + n_test)

else:
    if stratify is not None:
        CVClass = StratifiedShuffleSplit
    else:
        CVClass = ShuffleSplit

    cv = CVClass(test_size=n_test,
                  train_size=n_train,
                  random_state=random_state)

    train, test = next(cv.split(X=arrays[0], y=stratify))

return list(chain.from_iterable((_safe_indexing(a, train),
                                 _safe_indexing(a, test)) for a in arrays))

# Tell nose that train_test_split is not a test.
# (Needed for external libraries that may use nose.)
# Use setattr to avoid mypy errors when monkeypatching.
setattr(train_test_split, '__test__', False)

def _build_repr(self):
    # XXX This is copied from BaseEstimator's get_params
    cls = self.__class__
    init = getattr(cls.__init__, 'deprecated_original', cls.__init__)
    # Ignore varargs, kw and default values and pop self
    init_signature = signature(init)
    # Consider the constructor parameters excluding 'self'
    if init is object.__init__:
        args = []
    else:
        args = sorted([p.name for p in init_signature.parameters.values()
                      if p.name != 'self' and p.kind != p.VAR_KEYWORD])
    class_name = self.__class__.__name__
    params = dict()
    for key in args:
        # We need deprecation warnings to always be on in order to
        # catch deprecated param values.
        # This is set in utils/__init__.py but it gets overwritten
        # when running under python3 somehow.
        warnings.simplefilter("always", FutureWarning)
        try:
            with warnings.catch_warnings(record=True) as w:
                value = getattr(self, key, None)
                if value is None and hasattr(self, 'cvargs'):
                    value = self.cvargs.get(key, None)
            if len(w) and w[0].category == FutureWarning:
                # if the parameter is deprecated, don't show it
                continue
        finally:
            warnings.filters.pop(0)
            params[key] = value

    return '%s(%s)' % (class_name, _ pprint(params, offset=len(class_name)))

```

https://github.com/scikit-learn/scikit-learn/blob/master/sklearn/metrics/_classification.py

```
"""Metrics to assess performance on classification task given class prediction

Functions named as ``*_score`` return a scalar value to maximize: the higher
the better

Function named as ``*_error`` or ``*_loss`` return a scalar value to minimize:
the lower the better
"""

# Authors: Alexandre Gramfort <alexandre.gramfort@inria.fr>
#          Mathieu Blondel <mathieu@mblondel.org>
#          Olivier Grisel <olivier.grisel@ensta.org>
#          Arnaud Joly <a.joly@ulg.ac.be>
#          Jochen Wersdorfer <jochen@wersdoerfer.de>
#          Lars Buitinck
#          Joel Nothman <joel.nothman@gmail.com>
#          Noel Dawe <noel@dawe.me>
#          Jatin Shah <jatindshah@gmail.com>
#          Saurabh Jha <saurabh.jhaa@gmail.com>
#          Bernardo Stein <bernardovstein@gmail.com>
#          Shangwu Yao <shangwuyao@gmail.com>
# License: BSD 3 clause

import warnings
import numpy as np

from scipy.sparse import coo_matrix
from scipy.sparse import csr_matrix

from ..preprocessing import LabelBinarizer
from ..preprocessing import LabelEncoder
from ..utils import assert_all_finite
from ..utils import check_array
from ..utils import check_consistent_length
from ..utils import column_or_1d
from ..utils.multiclass import unique_labels
from ..utils.multiclass import type_of_target
from ..utils.validation import _num_samples
from ..utils.validation import _deprecate_positional_args
from ..sparsefuncs import count_nonzero
from ..exceptions import UndefinedMetricWarning

def _check_zero_division(zero_division):
    if isinstance(zero_division, str) and zero_division == "warn":
        return
    elif isinstance(zero_division, (int, float)) and zero_division in [0, 1]:
        return
    raise ValueError('Got zero_division={0}.'
                     ' Must be one of ["warn", 0, 1]'.format(zero_division))

def _check_targets(y_true, y_pred):
    """Check that y_true and y_pred belong to the same classification task
```

This converts multiclass or binary types to a common shape, and raises a ValueError for a mix of multilabel and multiclass targets, a mix of multilabel formats, for the presence of continuous-valued or multioutput targets, or for targets of different lengths.

Column vectors are squeezed to 1d, while multilabel formats are returned as CSR sparse label indicators.

Parameters

y_true : array-like

y_pred : array-like

Returns

type_true : one of {'multilabel-indicator', 'multiclass', 'binary'}

The type of the true target data, as output by
``utils.multiclass.type_of_target``

y_true : array or indicator matrix

y_pred : array or indicator matrix

"""

check_consistent_length(y_true, y_pred)

type_true = type_of_target(y_true)

type_pred = type_of_target(y_pred)

y_type = {type_true, type_pred}

if y_type == {"binary", "multiclass"}:
 y_type = {"multiclass"}

if len(y_type) > 1:
 raise ValueError("Classification metrics can't handle a mix of {0} "
 "and {1} targets".format(type_true, type_pred))

We can't have more than one value on y_type => The set is no more needed
y_type = y_type.pop()

No metrics support "multiclass-multioutput" format

if (y_type not in ["binary", "multiclass", "multilabel-indicator"]):
 raise ValueError("{0} is not supported".format(y_type))

if y_type in ["binary", "multiclass"]:

y_true = column_or_1d(y_true)

y_pred = column_or_1d(y_pred)

if y_type == "binary":

unique_values = np.union1d(y_true, y_pred)

if len(unique_values) > 2:

y_type = "multiclass"

if y_type.startswith('multilabel'):

y_true = csr_matrix(y_true)

y_pred = csr_matrix(y_pred)

y_type = 'multilabel-indicator'

return y_type, y_true, y_pred

def _weighted_sum(sample_score, sample_weight, normalize=False):

if normalize:

return np.average(sample_score, weights=sample_weight)

```
        elif sample_weight is not None:
            return np.dot(sample_score, sample_weight)
        else:
            return sample_score.sum()

 @_deprecate_positional_args
def accuracy_score(y_true, y_pred, *, normalize=True, sample_weight=None):
    """Accuracy classification score.

    In multilabel classification, this function computes subset accuracy:
    the set of labels predicted for a sample must *exactly* match the
    corresponding set of labels in y_true.

    Read more in the :ref:`User Guide <accuracy_score>`.

    Parameters
    -----
    y_true : 1d array-like, or label indicator array / sparse matrix
        Ground truth (correct) labels.

    y_pred : 1d array-like, or label indicator array / sparse matrix
        Predicted labels, as returned by a classifier.

    normalize : bool, optional (default=True)
        If ``False``, return the number of correctly classified samples.
        Otherwise, return the fraction of correctly classified samples.

    sample_weight : array-like of shape (n_samples,), default=None
        Sample weights.

    Returns
    -----
    score : float
        If ``normalize == True``, return the fraction of correctly
        classified samples (float), else returns the number of correctly
        classified samples (int).

        The best performance is 1 with ``normalize == True`` and the number
        of samples with ``normalize == False``.

    See also
    -----
    jaccard_score, hamming_loss, zero_one_loss

    Notes
    -----
    In binary and multiclass classification, this function is equal
    to the ``jaccard_score`` function.

    Examples
    -----
    >>> from sklearn.metrics import accuracy_score
    >>> y_pred = [0, 2, 1, 3]
    >>> y_true = [0, 1, 2, 3]
    >>> accuracy_score(y_true, y_pred)
    0.5
    >>> accuracy_score(y_true, y_pred, normalize=False)
    2

    In the multilabel case with binary label indicators:

    >>> import numpy as np
```

```

>>> accuracy_score(np.array([[0, 1], [1, 1]]), np.ones((2, 2)))
0.5
"""

# Compute accuracy for each possible representation
y_type, y_true, y_pred = _check_targets(y_true, y_pred)
check_consistent_length(y_true, y_pred, sample_weight)
if y_type.startswith('multilabel'):
    differing_labels = count_nonzero(y_true - y_pred, axis=1)
    score = differing_labels == 0
else:
    score = y_true == y_pred

return _weighted_sum(score, sample_weight, normalize)

 @_deprecate_positional_args
def confusion_matrix(y_true, y_pred, *, labels=None, sample_weight=None,
                      normalize=None):
    """Compute confusion matrix to evaluate the accuracy of a classification.

    By definition a confusion matrix :math:`C` is such that :math:`C_{i, j}` is equal to the number of observations known to be in group :math:`i` and predicted to be in group :math:`j`.

    Thus in binary classification, the count of true negatives is :math:`C_{0,0}`, false negatives is :math:`C_{1,0}`, true positives is :math:`C_{1,1}` and false positives is :math:`C_{0,1}`.

    Read more in the :ref:`User Guide <confusion_matrix>`.

    Parameters
    -----
    y_true : array-like of shape (n_samples,)
        Ground truth (correct) target values.

    y_pred : array-like of shape (n_samples,)
        Estimated targets as returned by a classifier.

    labels : array-like of shape (n_classes), default=None
        List of labels to index the matrix. This may be used to reorder or select a subset of labels.
        If ``None`` is given, those that appear at least once in ``y_true`` or ``y_pred`` are used in sorted order.

    sample_weight : array-like of shape (n_samples,), default=None
        Sample weights.

    normalize : {'true', 'pred', 'all'}, default=None
        Normalizes confusion matrix over the true (rows), predicted (columns) conditions or all the population. If None, confusion matrix will not be normalized.

    Returns
    -----
    C : ndarray of shape (n_classes, n_classes)
        Confusion matrix whose i-th row and j-th column entry indicates the number of samples with true label being i-th class and predicted label being j-th class.

    References
    -----

```

```

.. [1] `Wikipedia entry for the Confusion matrix
      <https://en.wikipedia.org/wiki/Confusion_matrix>`_
      (Wikipedia and other references may use a different
      convention for axes)

Examples
-----
>>> from sklearn.metrics import confusion_matrix
>>> y_true = [2, 0, 2, 2, 0, 1]
>>> y_pred = [0, 0, 2, 2, 0, 2]
>>> confusion_matrix(y_true, y_pred)
array([[2, 0, 0],
       [0, 0, 1],
       [1, 0, 2]])

>>> y_true = ["cat", "ant", "cat", "cat", "ant", "bird"]
>>> y_pred = ["ant", "ant", "cat", "cat", "ant", "cat"]
>>> confusion_matrix(y_true, y_pred, labels=["ant", "bird", "cat"])
array([[2, 0, 0],
       [0, 0, 1],
       [1, 0, 2]]]

In the binary case, we can extract true positives, etc as follows:

>>> tn, fp, fn, tp = confusion_matrix([0, 1, 0, 1], [1, 1, 1, 0]).ravel()
>>> (tn, fp, fn, tp)
(0, 2, 1, 1)

"""
y_type, y_true, y_pred = _check_targets(y_true, y_pred)
if y_type not in ("binary", "multiclass"):
    raise ValueError("%s is not supported" % y_type)

if labels is None:
    labels = unique_labels(y_true, y_pred)
else:
    labels = np.asarray(labels)
n_labels = labels.size
if n_labels == 0:
    raise ValueError("'labels' should contains at least one label.")
elif y_true.size == 0:
    return np.zeros((n_labels, n_labels), dtype=np.int)
elif np.all([l not in y_true for l in labels]):
    raise ValueError("At least one label specified must be in y_true")

if sample_weight is None:
    sample_weight = np.ones(y_true.shape[0], dtype=np.int64)
else:
    sample_weight = np.asarray(sample_weight)

check_consistent_length(y_true, y_pred, sample_weight)

if normalize not in ['true', 'pred', 'all', None]:
    raise ValueError("normalize must be one of {'true', 'pred', "
                     "'all', None}")

n_labels = labels.size
label_to_ind = {y: x for x, y in enumerate(labels)}
# convert yt, yp into index
y_pred = np.array([label_to_ind.get(x, n_labels + 1) for x in y_pred])
y_true = np.array([label_to_ind.get(x, n_labels + 1) for x in y_true])

# intersect y_pred, y_true with labels, eliminate items not in labels

```

```

ind = np.logical_and(y_pred < n_labels, y_true < n_labels)
y_pred = y_pred[ind]
y_true = y_true[ind]
# also eliminate weights of eliminated items
sample_weight = sample_weight[ind]

# Choose the accumulator dtype to always have high precision
if sample_weight.dtype.kind in {'i', 'u', 'b'}:
    dtype = np.int64
else:
    dtype = np.float64

cm = coo_matrix((sample_weight, (y_true, y_pred)),
                shape=(n_labels, n_labels), dtype=dtype,
                ).toarray()

with np.errstate(all='ignore'):
    if normalize == 'true':
        cm = cm / cm.sum(axis=1, keepdims=True)
    elif normalize == 'pred':
        cm = cm / cm.sum(axis=0, keepdims=True)
    elif normalize == 'all':
        cm = cm / cm.sum()
    cm = np.nan_to_num(cm)

return cm

 @_deprecate_positional_args
def multilabel_confusion_matrix(y_true, y_pred, *, sample_weight=None,
                                 labels=None, samplewise=False):
    """Compute a confusion matrix for each class or sample

.. versionadded:: 0.21

Compute class-wise (default) or sample-wise (samplewise=True) multilabel
confusion matrix to evaluate the accuracy of a classification, and output
confusion matrices for each class or sample.

In multilabel confusion matrix :math:`MCM` , the count of true negatives
is :math:`MCM_{:,0,0}` , false negatives is :math:`MCM_{:,1,0}` ,
true positives is :math:`MCM_{:,1,1}` and false positives is
:math:`MCM_{:,0,1}` .

Multiclass data will be treated as if binarized under a one-vs-rest
transformation. Returned confusion matrices will be in the order of
sorted unique labels in the union of (y_true, y_pred).

Read more in the :ref:`User Guide <multilabel_confusion_matrix>` .

Parameters
-----
y_true : 1d array-like, or label indicator array / sparse matrix
    of shape (n_samples, n_outputs) or (n_samples,)
    Ground truth (correct) target values.

y_pred : 1d array-like, or label indicator array / sparse matrix
    of shape (n_samples, n_outputs) or (n_samples,)
    Estimated targets as returned by a classifier

sample_weight : array-like of shape (n_samples,), default=None
    Sample weights

```

```
labels : array-like
    A list of classes or column indices to select some (or to force
    inclusion of classes absent from the data)

samplewise : bool, default=False
    In the multilabel case, this calculates a confusion matrix per sample
```

Returns

```
multi_confusion : array, shape (n_outputs, 2, 2)
    A 2x2 confusion matrix corresponding to each output in the input.
    When calculating class-wise multi_confusion (default), then
    n_outputs = n_labels; when calculating sample-wise multi_confusion
    (samplewise=True), n_outputs = n_samples. If ``labels`` is defined,
    the results will be returned in the order specified in ``labels``,
    otherwise the results will be returned in sorted order by default.
```

See also

```
confusion_matrix
```

Notes

The multilabel_confusion_matrix calculates class-wise or sample-wise multilabel confusion matrices, and in multiclass tasks, labels are binarized under a one-vs-rest way; while confusion_matrix calculates one confusion matrix for confusion between every two classes.

Examples

Multilabel-indicator case:

```
>>> import numpy as np
>>> from sklearn.metrics import multilabel_confusion_matrix
>>> y_true = np.array([[1, 0, 1],
...                   [0, 1, 0]])
>>> y_pred = np.array([[1, 0, 0],
...                   [0, 1, 1]])
>>> multilabel_confusion_matrix(y_true, y_pred)
array([[[1, 0],
       [0, 1]],
<BLANKLINE>
      [[1, 0],
       [0, 1]],
<BLANKLINE>
      [[0, 1],
       [1, 0]])
```

Multiclass case:

```
>>> y_true = ["cat", "ant", "cat", "cat", "ant", "bird"]
>>> y_pred = ["ant", "ant", "cat", "cat", "ant", "cat"]
>>> multilabel_confusion_matrix(y_true, y_pred,
...                               labels=["ant", "bird", "cat"])
array([[[3, 1],
       [0, 2]],
<BLANKLINE>
      [[5, 0],
       [1, 0]],
<BLANKLINE>
      [[2, 1],
       [1, 2]])
```

```

"""
y_type, y_true, y_pred = _check_targets(y_true, y_pred)
if sample_weight is not None:
    sample_weight = column_or_1d(sample_weight)
check_consistent_length(y_true, y_pred, sample_weight)

if y_type not in ("binary", "multiclass", "multilabel-indicator"):
    raise ValueError("%s is not supported" % y_type)

present_labels = unique_labels(y_true, y_pred)
if labels is None:
    labels = present_labels
    n_labels = None
else:
    n_labels = len(labels)
    labels = np.hstack([labels, np.setdiff1d(present_labels, labels,
                                              assume_unique=True)])
if y_true.ndim == 1:
    if samplewise:
        raise ValueError("Samplewise metrics are not available outside of "
                         "multilabel classification.")

    le = LabelEncoder()
    le.fit(labels)
    y_true = le.transform(y_true)
    y_pred = le.transform(y_pred)
    sorted_labels = le.classes_

    # labels are now from 0 to len(labels) - 1 -> use bincount
    tp = y_true == y_pred
    tp_bins = y_true[tp]
    if sample_weight is not None:
        tp_bins_weights = np.asarray(sample_weight)[tp]
    else:
        tp_bins_weights = None

    if len(tp_bins):
        tp_sum = np.bincount(tp_bins, weights=tp_bins_weights,
                             minlength=len(labels))
    else:
        # Pathological case
        true_sum = pred_sum = tp_sum = np.zeros(len(labels))
    if len(y_pred):
        pred_sum = np.bincount(y_pred, weights=sample_weight,
                               minlength=len(labels))
    if len(y_true):
        true_sum = np.bincount(y_true, weights=sample_weight,
                               minlength=len(labels))

    # Retain only selected labels
    indices = np.searchsorted(sorted_labels, labels[:n_labels])
    tp_sum = tp_sum[indices]
    true_sum = true_sum[indices]
    pred_sum = pred_sum[indices]

else:
    sum_axis = 1 if samplewise else 0

    # All labels are index integers for multilabel.
    # Select labels:
    if not np.array_equal(labels, present_labels):

```

```

    if np.max(labels) > np.max(present_labels):
        raise ValueError('All labels must be in [0, n_labels) for '
                         'multilabel targets. '
                         'Got %d > %d' %
                         (np.max(labels), np.max(present_labels)))
    if np.min(labels) < 0:
        raise ValueError('All labels must be in [0, n_labels) for '
                         'multilabel targets. '
                         'Got %d < 0' % np.min(labels))

    if n_labels is not None:
        y_true = y_true[:, labels[:n_labels]]
        y_pred = y_pred[:, labels[:n_labels]]

    # calculate weighted counts
    true_and_pred = y_true.multiply(y_pred)
    tp_sum = count_nonzero(true_and_pred, axis=sum_axis,
                           sample_weight=sample_weight)
    pred_sum = count_nonzero(y_pred, axis=sum_axis,
                           sample_weight=sample_weight)
    true_sum = count_nonzero(y_true, axis=sum_axis,
                           sample_weight=sample_weight)

    fp = pred_sum - tp_sum
    fn = true_sum - tp_sum
    tp = tp_sum

    if sample_weight is not None and samplewise:
        sample_weight = np.array(sample_weight)
        tp = np.array(tp)
        fp = np.array(fp)
        fn = np.array(fn)
        tn = sample_weight * y_true.shape[1] - tp - fp - fn
    elif sample_weight is not None:
        tn = sum(sample_weight) - tp - fp - fn
    elif samplewise:
        tn = y_true.shape[1] - tp - fp - fn
    else:
        tn = y_true.shape[0] - tp - fp - fn

    return np.array([tn, fp, fn, tp]).T.reshape(-1, 2, 2)

```

`@_deprecate_positional_args`

```

def cohen_kappa_score(y1, y2, *, labels=None, weights=None,
                      sample_weight=None):
    """Cohen's kappa: a statistic that measures inter-annotator agreement.
```

This function computes Cohen's kappa [1]_, a score that expresses the level of agreement between two annotators on a classification problem. It is defined as

$$\kappa = (p_o - p_e) / (1 - p_e)$$

where :math:`p_o` is the empirical probability of agreement on the label assigned to any sample (the observed agreement ratio), and :math:`p_e` is the expected agreement when both annotators assign labels randomly. :math:`p_e` is estimated using a per-annotator empirical prior over the class labels [2]_.

Read more in the :ref:`User Guide <cohen_kappa>`.

Parameters

y1 : array, shape = [n_samples]
 Labels assigned by the first annotator.

y2 : array, shape = [n_samples]
 Labels assigned by the second annotator. The kappa statistic is symmetric, so swapping ``y1`` and ``y2`` doesn't change the value.

labels : array, shape = [n_classes], optional
 List of labels to index the matrix. This may be used to select a subset of labels. If None, all labels that appear at least once in ``y1`` or ``y2`` are used.

weights : str, optional
 Weighting type to calculate the score. None means no weighted; "linear" means linear weighted; "quadratic" means quadratic weighted.

sample_weight : array-like of shape (n_samples,), default=None
 Sample weights.

Returns

kappa : float
 The kappa statistic, which is a number between -1 and 1. The maximum value means complete agreement; zero or lower means chance agreement.

References

.. [1] J. Cohen (1960). "A coefficient of agreement for nominal scales".
 Educational and Psychological Measurement 20(1):37-46.
 doi:10.1177/001316446002000104.

.. [2] R. Artstein and M. Poesio (2008). "Inter-coder agreement for computational linguistics". Computational Linguistics 34(4):555-596.
 <<https://www.mitpressjournals.org/doi/pdf/10.1162/coli.07-034-R2>>`_

.. [3] Wikipedia entry for the Cohen's kappa.
 <https://en.wikipedia.org/wiki/Cohen%27s_kappa>`_

"""

```

confusion = confusion_matrix(y1, y2, labels=labels,
                             sample_weight=sample_weight)
n_classes = confusion.shape[0]
sum0 = np.sum(confusion, axis=0)
sum1 = np.sum(confusion, axis=1)
expected = np.outer(sum0, sum1) / np.sum(sum0)

if weights is None:
    w_mat = np.ones([n_classes, n_classes], dtype=np.int)
    w_mat.flat[::- n_classes + 1] = 0
elif weights == "linear" or weights == "quadratic":
    w_mat = np.zeros([n_classes, n_classes], dtype=np.int)
    w_mat += np.arange(n_classes)
    if weights == "linear":
        w_mat = np.abs(w_mat - w_mat.T)
    else:
        w_mat = (w_mat - w_mat.T) ** 2
else:
    raise ValueError("Unknown kappa weighting type.")

k = np.sum(w_mat * confusion) / np.sum(w_mat * expected)
return 1 - k

```

@_deprecate_positional_args

```

def jaccard_score(y_true, y_pred, *, labels=None, pos_label=1,
                  average='binary', sample_weight=None):
    """Jaccard similarity coefficient score

    The Jaccard index [1], or Jaccard similarity coefficient, defined as
    the size of the intersection divided by the size of the union of two label
    sets, is used to compare set of predicted labels for a sample to the
    corresponding set of labels in ``y_true``.

    Read more in the :ref:`User Guide <jaccard_similarity_score>`.

    Parameters
    -----
    y_true : 1d array-like, or label indicator array / sparse matrix
        Ground truth (correct) labels.

    y_pred : 1d array-like, or label indicator array / sparse matrix
        Predicted labels, as returned by a classifier.

    labels : list, optional
        The set of labels to include when ``average != 'binary'``, and their
        order if ``average is None``. Labels present in the data can be
        excluded, for example to calculate a multiclass average ignoring a
        majority negative class, while labels not present in the data will
        result in 0 components in a macro average. For multilabel targets,
        labels are column indices. By default, all labels in ``y_true`` and
        ``y_pred`` are used in sorted order.

    pos_label : str or int, 1 by default
        The class to report if ``average='binary'`` and the data is binary.
        If the data are multiclass or multilabel, this will be ignored;
        setting ``labels=[pos_label]`` and ``average != 'binary'`` will report
        scores for that label only.

    average : string, [None, 'binary' (default), 'micro', 'macro', 'samples', \
                  'weighted']
        If ``None``, the scores for each class are returned. Otherwise, this
        determines the type of averaging performed on the data:

        ``'binary'``:
            Only report results for the class specified by ``pos_label``.
            This is applicable only if targets (``y_{true,pred}``) are binary.

        ``'micro'``:
            Calculate metrics globally by counting the total true positives,
            false negatives and false positives.

        ``'macro'``:
            Calculate metrics for each label, and find their unweighted
            mean. This does not take label imbalance into account.

        ``'weighted'``:
            Calculate metrics for each label, and find their average, weighted
            by support (the number of true instances for each label). This
            alters 'macro' to account for label imbalance.

        ``'samples'``:
            Calculate metrics for each instance, and find their average (only
            meaningful for multilabel classification).

    sample_weight : array-like of shape (n_samples,), default=None
        Sample weights.

    Returns
    -----
    score : float (if average is not None) or array of floats, shape =\
        [n_unique_labels]

```

See also

accuracy_score, f_score, multilabel_confusion_matrix

Notes

:func:`jaccard_score` may be a poor metric if there are no positives for some samples or classes. Jaccard is undefined if there are no true or predicted labels, and our implementation will return a score of 0 with a warning.

References

[1] Wikipedia entry for the Jaccard index
https://en.wikipedia.org/wiki/Jaccard_index

Examples

```
>>> import numpy as np
>>> from sklearn.metrics import jaccard_score
>>> y_true = np.array([[0, 1, 1],
...                   [1, 1, 0]])
>>> y_pred = np.array([[1, 1, 1],
...                   [1, 0, 0]])
```

In the binary case:

```
>>> jaccard_score(y_true[0], y_pred[0])
0.6666...
```

In the multilabel case:

```
>>> jaccard_score(y_true, y_pred, average='samples')
0.5833...
>>> jaccard_score(y_true, y_pred, average='macro')
0.6666...
>>> jaccard_score(y_true, y_pred, average=None)
array([0.5, 0.5, 1.])
```

In the multiclass case:

```
>>> y_pred = [0, 2, 1, 2]
>>> y_true = [0, 1, 2, 2]
>>> jaccard_score(y_true, y_pred, average=None)
array([1., 0., 0.33...])
"""
labels = _check_set_wise_labels(y_true, y_pred, average, labels,
                                pos_label)
samplewise = average == 'samples'
MCM = multilabel_confusion_matrix(y_true, y_pred,
                                   sample_weight=sample_weight,
                                   labels=labels, samplewise=samplewise)
numerator = MCM[:, 1, 1]
denominator = MCM[:, 1, 1] + MCM[:, 0, 1] + MCM[:, 1, 0]

if average == 'micro':
    numerator = np.array([numerator.sum()])
    denominator = np.array([denominator.sum()])

jaccard = _prf_divide(numerator, denominator, 'jaccard',
                      'true or predicted', average, ('jaccard',))
if average is None:
```

```
    return jaccard
if average == 'weighted':
    weights = MCM[:, 1, 0] + MCM[:, 1, 1]
    if not np.any(weights):
        # numerator is 0, and warning should have already been issued
        weights = None
elif average == 'samples' and sample_weight is not None:
    weights = sample_weight
else:
    weights = None
return np.average(jaccard, weights=weights)
```

```
@_deprecate_positional_args
def matthews_corrcoef(y_true, y_pred, *, sample_weight=None):
    """Compute the Matthews correlation coefficient (MCC)
```

The Matthews correlation coefficient is used in machine learning as a measure of the quality of binary and multiclass classifications. It takes into account true and false positives and negatives and is generally regarded as a balanced measure which can be used even if the classes are of very different sizes. The MCC is in essence a correlation coefficient value between -1 and +1. A coefficient of +1 represents a perfect prediction, 0 an average random prediction and -1 an inverse prediction. The statistic is also known as the phi coefficient. [source: Wikipedia]

Binary and multiclass labels are supported. Only in the binary case does this relate to information about true and false positives and negatives. See references below.

Read more in the :ref:`User Guide <matthews_corrcoef>`.

Parameters

y_true : array, shape = [n_samples]
Ground truth (correct) target values.

y_pred : array, shape = [n_samples]
Estimated targets as returned by a classifier.

sample_weight : array-like of shape (n_samples,), default=None
Sample weights.

Returns

mcc : float
The Matthews correlation coefficient (+1 represents a perfect prediction, 0 an average random prediction and -1 an inverse prediction).

References

.. [1] `Baldi, Brunak, Chauvin, Andersen and Nielsen, (2000). Assessing the accuracy of prediction algorithms for classification: an overview <<https://doi.org/10.1093/bioinformatics/16.5.412>>`_

.. [2] `Wikipedia entry for the Matthews Correlation Coefficient <https://en.wikipedia.org/wiki/Matthews_correlation_coefficient>`_

.. [3] `Gorodkin, (2004). Comparing two K-category assignments by a K-category correlation coefficient <<https://www.sciencedirect.com/science/article/pii/S1476927104000799>>`_

```
.. [4] `Jurman, Riccadonna, Furlanello, (2012). A Comparison of MCC and CEN
      Error Measures in MultiClass Prediction
      <https://journals.plos.org/plosone/article?id=10.1371/journal.pone.0041882>`_
```

Examples

```
>>> from sklearn.metrics import matthews_corrcoef
>>> y_true = [+1, +1, +1, -1]
>>> y_pred = [+1, -1, +1, +1]
>>> matthews_corrcoef(y_true, y_pred)
-0.33...
"""
y_type, y_true, y_pred = _check_targets(y_true, y_pred)
check_consistent_length(y_true, y_pred, sample_weight)
if y_type not in {"binary", "multiclass"}:
    raise ValueError("%s is not supported" % y_type)

lb = LabelEncoder()
lb.fit(np.hstack([y_true, y_pred]))
y_true = lb.transform(y_true)
y_pred = lb.transform(y_pred)

C = confusion_matrix(y_true, y_pred, sample_weight=sample_weight)
t_sum = C.sum(axis=1, dtype=np.float64)
p_sum = C.sum(axis=0, dtype=np.float64)
n_correct = np.trace(C, dtype=np.float64)
n_samples = p_sum.sum()
cov_ytyp = n_correct * n_samples - np.dot(t_sum, p_sum)
cov_ypyp = n_samples ** 2 - np.dot(p_sum, p_sum)
cov_ytyt = n_samples ** 2 - np.dot(t_sum, t_sum)
mcc = cov_ytyp / np.sqrt(cov_ytyt * cov_ypyp)

if np.isnan(mcc):
    return 0.
else:
    return mcc
```

@_deprecate_positional_args

```
def zero_one_loss(y_true, y_pred, *, normalize=True, sample_weight=None):
    """Zero-one classification loss.
```

If normalize is ``True``, return the fraction of misclassifications (float), else it returns the number of misclassifications (int). The best performance is 0.

Read more in the :ref:`User Guide <zero_one_loss>`.

Parameters

y_true : 1d array-like, or label indicator array / sparse matrix
Ground truth (correct) labels.

y_pred : 1d array-like, or label indicator array / sparse matrix
Predicted labels, as returned by a classifier.

normalize : bool, optional (default=True)
If ``False``, return the number of misclassifications.
Otherwise, return the fraction of misclassifications.

sample_weight : array-like of shape (n_samples,), default=None
Sample weights.

```
Returns
-----
loss : float or int,
    If ``normalize == True``, return the fraction of misclassifications
    (float), else it returns the number of misclassifications (int).

Notes
-----
In multilabel classification, the zero_one_loss function corresponds to
the subset zero-one loss: for each sample, the entire set of labels must be
correctly predicted, otherwise the loss for that sample is equal to one.

See also
-----
accuracy_score, hamming_loss, jaccard_score
```

```
Examples
-----
>>> from sklearn.metrics import zero_one_loss
>>> y_pred = [1, 2, 3, 4]
>>> y_true = [2, 2, 3, 4]
>>> zero_one_loss(y_true, y_pred)
0.25
>>> zero_one_loss(y_true, y_pred, normalize=False)
1
```

In the multilabel case with binary label indicators:

```
>>> import numpy as np
>>> zero_one_loss(np.array([[0, 1], [1, 1]]), np.ones((2, 2)))
0.5
"""
score = accuracy_score(y_true, y_pred,
                       normalize=normalize,
                       sample_weight=sample_weight)

if normalize:
    return 1 - score
else:
    if sample_weight is not None:
        n_samples = np.sum(sample_weight)
    else:
        n_samples = _num_samples(y_true)
    return n_samples - score

 @_deprecate_positional_args
def f1_score(y_true, y_pred, *, labels=None, pos_label=1, average='binary',
             sample_weight=None, zero_division="warn"):
    """Compute the F1 score, also known as balanced F-score or F-measure
```

The F1 score can be interpreted as a weighted average of the precision and recall, where an F1 score reaches its best value at 1 and worst score at 0. The relative contribution of precision and recall to the F1 score are equal. The formula for the F1 score is::

$$F1 = 2 * (\text{precision} * \text{recall}) / (\text{precision} + \text{recall})$$

In the multi-class and multi-label case, this is the average of the F1 score of each class with weighting depending on the ``average`` parameter.

Read more in the :ref:`User Guide <precision_recall_f_measure_metrics>`.

Parameters

y_true : 1d array-like, or label indicator array / sparse matrix
 Ground truth (correct) target values.

y_pred : 1d array-like, or label indicator array / sparse matrix
 Estimated targets as returned by a classifier.

labels : list, optional
 The set of labels to include when ``average != 'binary'``, and their
 order if ``average is None``. Labels present in the data can be
 excluded, for example to calculate a multiclass average ignoring a
 majority negative class, while labels not present in the data will
 result in 0 components in a macro average. For multilabel targets,
 labels are column indices. By default, all labels in ``y_true`` and
 ``y_pred`` are used in sorted order.

.. versionchanged:: 0.17
 parameter *labels* improved for multiclass problem.

pos_label : str or int, 1 by default
 The class to report if ``average='binary'`` and the data is binary.
 If the data are multiclass or multilabel, this will be ignored;
 setting ``labels=[pos_label]`` and ``average != 'binary'`` will report
 scores for that label only.

average : string, [None, 'binary' (default), 'micro', 'macro', 'samples', '
 'weighted']
 This parameter is required for multiclass/multilabel targets.
 If ``None``, the scores for each class are returned. Otherwise, this
 determines the type of averaging performed on the data:

``'binary'``:
 Only report results for the class specified by ``pos_label``.
 This is applicable only if targets (``y_{true,pred}``) are binary.

``'micro'``:
 Calculate metrics globally by counting the total true positives,
 false negatives and false positives.

``'macro'``:
 Calculate metrics for each label, and find their unweighted
 mean. This does not take label imbalance into account.

``'weighted'``:
 Calculate metrics for each label, and find their average weighted
 by support (the number of true instances for each label). This
 alters 'macro' to account for label imbalance; it can result in an
 F-score that is not between precision and recall.

``'samples'``:
 Calculate metrics for each instance, and find their average (only
 meaningful for multilabel classification where this differs from
 :func:`accuracy_score`).

sample_weight : array-like of shape (n_samples,), default=None
 Sample weights.

zero_division : "warn", 0 or 1, default="warn"
 Sets the value to return when there is a zero division, i.e. when all
 predictions and labels are negative. If set to "warn", this acts as 0,
 but warnings are also raised.

Returns

f1_score : float or array of float, shape = [n_unique_labels]

F1 score of the positive class in binary classification or weighted average of the F1 scores of each class for the multiclass task.

See also

fbeta_score, precision_recall_fscore_support, jaccard_score,
multilabel_confusion_matrix

References

.. [1] `Wikipedia entry for the F1-score
<https://en.wikipedia.org/wiki/F1_score>`_

Examples

>>> from sklearn.metrics import f1_score
>>> y_true = [0, 1, 2, 0, 1, 2]
>>> y_pred = [0, 2, 1, 0, 0, 1]
>>> f1_score(y_true, y_pred, average='macro')
0.26...
>>> f1_score(y_true, y_pred, average='micro')
0.33...
>>> f1_score(y_true, y_pred, average='weighted')
0.26...
>>> f1_score(y_true, y_pred, average=None)
array([0.8, 0., 0.])
>>> y_true = [0, 0, 0, 0, 0, 0]
>>> y_pred = [0, 0, 0, 0, 0, 0]
>>> f1_score(y_true, y_pred, zero_division=1)
1.0...
1.0...

Notes

When ``true positive + false positive == 0``, precision is undefined.
When ``true positive + false negative == 0``, recall is undefined.
In such cases, by default the metric will be set to 0, as will f-score,
and ``UndefinedMetricWarning`` will be raised. This behavior can be
modified with ``zero_division``.

"""

```
return fbeta_score(y_true, y_pred, beta=1, labels=labels,  
                   pos_label=pos_label, average=average,  
                   sample_weight=sample_weight,  
                   zero_division=zero_division)
```

@_deprecate_positional_args

```
def fbeta_score(y_true, y_pred, *, beta, labels=None, pos_label=1,  
                average='binary', sample_weight=None, zero_division="warn"):  
    """Compute the F-beta score
```

The F-beta score is the weighted harmonic mean of precision and recall,
reaching its optimal value at 1 and its worst value at 0.

The `beta` parameter determines the weight of recall in the combined
score. ``beta < 1`` lends more weight to precision, while ``beta > 1``
favors recall (``beta -> 0`` considers only precision, ``beta -> +inf``
only recall).

Read more in the :ref:`User Guide <precision_recall_f_measure_metrics>`.

Parameters

`y_true` : 1d array-like, or label indicator array / sparse matrix

Ground truth (correct) target values.

y_pred : 1d array-like, or label indicator array / sparse matrix
 Estimated targets as returned by a classifier.

beta : float
 Determines the weight of recall in the combined score.

labels : list, optional
 The set of labels to include when ``average != 'binary'``, and their order if ``average is None``. Labels present in the data can be excluded, for example to calculate a multiclass average ignoring a majority negative class, while labels not present in the data will result in 0 components in a macro average. For multilabel targets, labels are column indices. By default, all labels in ``y_true`` and ``y_pred`` are used in sorted order.

.. versionchanged:: 0.17
 parameter *labels* improved for multiclass problem.

pos_label : str or int, 1 by default
 The class to report if ``average='binary'`` and the data is binary.
 If the data are multiclass or multilabel, this will be ignored;
 setting ``labels=[pos_label]`` and ``average != 'binary'`` will report scores for that label only.

average : string, [None, 'binary' (default), 'micro', 'macro', 'samples', \
 'weighted']
 This parameter is required for multiclass/multilabel targets.
 If ``None``, the scores for each class are returned. Otherwise, this determines the type of averaging performed on the data:

```
```'binary'``:  

 Only report results for the class specified by ``pos_label``.

 This is applicable only if targets (``y_{true,pred}``) are binary.

```'micro'``:  

    Calculate metrics globally by counting the total true positives,  

    false negatives and false positives.  

```'macro'``:  

 Calculate metrics for each label, and find their unweighted

 mean. This does not take label imbalance into account.

```'weighted'``:  

    Calculate metrics for each label, and find their average weighted  

    by support (the number of true instances for each label). This  

    alters 'macro' to account for label imbalance; it can result in an  

    F-score that is not between precision and recall.  

```'samples'``:  

 Calculate metrics for each instance, and find their average (only

 meaningful for multilabel classification where this differs from

 :func:`accuracy_score`).
```

sample\_weight : array-like of shape (n\_samples,), default=None  
     Sample weights.

zero\_division : "warn", 0 or 1, default="warn"  
     Sets the value to return when there is a zero division, i.e. when all predictions and labels are negative. If set to "warn", this acts as 0, but warnings are also raised.

Returns

-----

fbeta\_score : float (if average is not None) or array of float, shape =\n     [n\_unique\_labels]

F-beta score of the positive class in binary classification or weighted average of the F-beta score of each class for the multiclass task.

See also

precision\_recall\_fscore\_support, multilabel\_confusion\_matrix

References

- .. [1] R. Baeza-Yates and B. Ribeiro-Neto (2011).  
Modern Information Retrieval. Addison Wesley, pp. 327-328.
- .. [2] `Wikipedia entry for the F1-score  
<[>`\\_](https://en.wikipedia.org/wiki/F1_score)

Examples

```
>>> from sklearn.metrics import fbeta_score
>>> y_true = [0, 1, 2, 0, 1, 2]
>>> y_pred = [0, 2, 1, 0, 0, 1]
>>> fbeta_score(y_true, y_pred, average='macro', beta=0.5)
0.23...
>>> fbeta_score(y_true, y_pred, average='micro', beta=0.5)
0.33...
>>> fbeta_score(y_true, y_pred, average='weighted', beta=0.5)
0.23...
>>> fbeta_score(y_true, y_pred, average=None, beta=0.5)
array([0.71..., 0. , 0.])
```

Notes

When ``true positive + false positive == 0`` or  
``true positive + false negative == 0```, f-score returns 0 and raises  
``UndefinedMetricWarning``. This behavior can be  
modified with ``zero\_division``.

"""

```
,,_ = precision_recall_fscore_support(y_true, y_pred,
 beta=beta,
 labels=labels,
 pos_label=pos_label,
 average=average,
 warn_for=('f-score',),
 sample_weight=sample_weight,
 zero_division=zero_division)

return f
```

```
def _prf_divide(numerator, denominator, metric,
 modifier, average, warn_for, zero_division="warn"):
 """Performs division and handles divide-by-zero.
```

On zero-division, sets the corresponding result elements equal to  
0 or 1 (according to ``zero\_division``). Plus, if  
``zero\_division != "warn"`` raises a warning.

The metric, modifier and average arguments are used only for determining  
an appropriate warning.

"""

```
mask = denominator == 0.0
denominator = denominator.copy()
denominator[mask] = 1 # avoid infs/nans
result = numerator / denominator
```

```

if not np.any(mask):
 return result

if ``zero_division=1``, set those with denominator == 0 equal to 1
result[mask] = 0.0 if zero_division in ["warn", 0] else 1.0

the user will be removing warnings if zero_division is set to something
different than its default value. If we are computing only f-score
the warning will be raised only if precision and recall are ill-defined
if zero_division != "warn" or metric not in warn_for:
 return result

build appropriate warning
E.g. "Precision and F-score are ill-defined and being set to 0.0 in
labels with no predicted samples. Use ``zero_division`` parameter to
control this behavior."

if metric in warn_for and 'f-score' in warn_for:
 msg_start = '{0} and F-score are'.format(metric.title())
elif metric in warn_for:
 msg_start = '{0}'.format(metric.title())
elif 'f-score' in warn_for:
 msg_start = 'F-score is'
else:
 return result

_warn_prf(average, modifier, msg_start, len(result))

return result

def _warn_prf(average, modifier, msg_start, result_size):
 axis0, axis1 = 'sample', 'label'
 if average == 'samples':
 axis0, axis1 = axis1, axis0
 msg = ('{0} ill-defined and being set to 0.0 {{0}} '
 'no {1} {2}s. Use `zero_division` parameter to control'
 ' this behavior.'.format(msg_start, modifier, axis0))
 if result_size == 1:
 msg = msg.format('due to')
 else:
 msg = msg.format('in {0}s with'.format(axis1))
 warnings.warn(msg, UndefinedMetricWarning, stacklevel=2)

def _check_set_wise_labels(y_true, y_pred, average, labels, pos_label):
 """Validation associated with set-wise metrics

 Returns identified labels
 """
 average_options = (None, 'micro', 'macro', 'weighted', 'samples')
 if average not in average_options and average != 'binary':
 raise ValueError('average has to be one of ' +
 str(average_options))

 y_type, y_true, y_pred = _check_targets(y_true, y_pred)
 present_labels = unique_labels(y_true, y_pred)
 if average == 'binary':
 if y_type == 'binary':
 if pos_label not in present_labels:
 if len(present_labels) >= 2:
 raise ValueError("pos_label=%r is not a valid label: "
```
```

```

        "%r" % (pos_label, present_labels))
    labels = [pos_label]
else:
    average_options = list(average_options)
    if y_type == 'multiclass':
        average_options.remove('samples')
    raise ValueError("Target is %s but average='binary'. Please "
                    "choose another average setting, one of %r."
                    % (y_type, average_options))
elif pos_label not in (None, 1):
    warnings.warn("Note that pos_label (set to %r) is ignored when "
                  "average != 'binary' (got %r). You may use "
                  "labels=[pos_label] to specify a single positive class."
                  % (pos_label, average), UserWarning)
return labels

 @_deprecate_positional_args
def precision_recall_fscore_support(y_true, y_pred, *, beta=1.0, labels=None,
                                      pos_label=1, average=None,
                                      warn_for=('precision', 'recall',
                                                'f-score'),
                                      sample_weight=None,
                                      zero_division="warn"):
    """Compute precision, recall, F-measure and support for each class

```

The precision is the ratio `` $tp / (tp + fp)$ `` where `` tp `` is the number of true positives and `` fp `` the number of false positives. The precision is intuitively the ability of the classifier not to label as positive a sample that is negative.

The recall is the ratio `` $tp / (tp + fn)$ `` where `` tp `` is the number of true positives and `` fn `` the number of false negatives. The recall is intuitively the ability of the classifier to find all the positive samples.

The F-beta score can be interpreted as a weighted harmonic mean of the precision and recall, where an F-beta score reaches its best value at 1 and worst score at 0.

The F-beta score weights recall more than precision by a factor of `` β ``. `` $\beta = 1.0$ `` means recall and precision are equally important.

The support is the number of occurrences of each class in `` y_{true} ``.

If `` pos_label is `None``` and in binary classification, this function returns the average precision, recall and F-measure if ```average``` is one of ``'micro'``, ``'macro'``, ``'weighted'`` or ``'samples'``.

Read more in the :ref:`User Guide <precision_recall_f_measure_metrics>`.

Parameters

`y_true` : 1d array-like, or label indicator array / sparse matrix
Ground truth (correct) target values.

`y_pred` : 1d array-like, or label indicator array / sparse matrix
Estimated targets as returned by a classifier.

`beta` : float, 1.0 by default
The strength of recall versus precision in the F-score.

`labels` : list, optional
The set of labels to include when ```average != 'binary'```, and their

order if ``average is None``. Labels present in the data can be excluded, for example to calculate a multiclass average ignoring a majority negative class, while labels not present in the data will result in 0 components in a macro average. For multilabel targets, labels are column indices. By default, all labels in ``y_true`` and ``y_pred`` are used in sorted order.

pos_label : str or int, 1 by default
The class to report if ``average='binary'`` and the data is binary.
If the data are multiclass or multilabel, this will be ignored;
setting ``labels=[pos_label]`` and ``average != 'binary'`` will report scores for that label only.

average : string, [None (default), 'binary', 'micro', 'macro', 'samples', '
'weighted']
If ``None``, the scores for each class are returned. Otherwise, this determines the type of averaging performed on the data:

```binary```:  
Only report results for the class specified by ``pos\_label``.  
This is applicable only if targets (``y\_{true,pred}``) are binary.  
```micro```:  
Calculate metrics globally by counting the total true positives,
false negatives and false positives.
```macro```:  
Calculate metrics for each label, and find their unweighted mean.  
This does not take label imbalance into account.  
```weighted```:  
Calculate metrics for each label, and find their average weighted by support (the number of true instances for each label). This alters 'macro' to account for label imbalance; it can result in an F-score that is not between precision and recall.
```samples```:  
Calculate metrics for each instance, and find their average (only meaningful for multilabel classification where this differs from :func:`accuracy\_score`).

warn\_for : tuple or set, for internal use  
This determines which warnings will be made in the case that this function is being used to return only one of its metrics.

sample\_weight : array-like of shape (n\_samples,), default=None  
Sample weights.

zero\_division : "warn", 0 or 1, default="warn"  
Sets the value to return when there is a zero division:  
- recall: when there are no positive labels  
- precision: when there are no positive predictions  
- f-score: both

If set to "warn", this acts as 0, but warnings are also raised.

Returns

-----

precision : float (if average is not None) or array of float, shape =\n[n\_unique\_labels]

recall : float (if average is not None) or array of float, , shape =\n[n\_unique\_labels]

fbeta\_score : float (if average is not None) or array of float, shape =\n[n\_unique\_labels]

```

support : None (if average is not None) or array of int, shape =\
[n_unique_labels]
The number of occurrences of each label in ``y_true``.

References

.. [1] `Wikipedia entry for the Precision and recall
<https://en.wikipedia.org/wiki/Precision_and_recall>`_
.. [2] `Wikipedia entry for the F1-score
<https://en.wikipedia.org/wiki/F1_score>`_
.. [3] `Discriminative Methods for Multi-labeled Classification Advances
in Knowledge Discovery and Data Mining (2004), pp. 22-30 by Shantanu
Godbole, Sunita Sarawagi
<http://www.godbole.net/shantanu/pubs/multilabelsvm-pakdd04.pdf>`_

```

## Examples

```

>>> import numpy as np
>>> from sklearn.metrics import precision_recall_fscore_support
>>> y_true = np.array(['cat', 'dog', 'pig', 'cat', 'dog', 'pig'])
>>> y_pred = np.array(['cat', 'pig', 'dog', 'cat', 'cat', 'dog'])
>>> precision_recall_fscore_support(y_true, y_pred, average='macro')
(0.22..., 0.33..., 0.26..., None)
>>> precision_recall_fscore_support(y_true, y_pred, average='micro')
(0.33..., 0.33..., 0.33..., None)
>>> precision_recall_fscore_support(y_true, y_pred, average='weighted')
(0.22..., 0.33..., 0.26..., None)

```

It is possible to compute per-label precisions, recalls, F1-scores and supports instead of averaging:

```

>>> precision_recall_fscore_support(y_true, y_pred, average=None,
... labels=['pig', 'dog', 'cat'])
(array([0. , 0. , 0.66...]),
 array([0., 0., 1.]), array([0., 0., 0.8]),
 array([2, 2, 2]))

```

## Notes

```

When ``true positive + false positive == 0``, precision is undefined;
When ``true positive + false negative == 0``, recall is undefined.
In such cases, by default the metric will be set to 0, as will f-score,
and ``UndefinedMetricWarning`` will be raised. This behavior can be
modified with ``zero_division``.
"""

```

```

_check_zero_division(zero_division)
if beta < 0:
 raise ValueError("beta should be >=0 in the F-beta score")
labels = _check_set_wise_labels(y_true, y_pred, average, labels,
 pos_label)

Calculate tp_sum, pred_sum, true_sum
samplewise = average == 'samples'
MCM = multilabel_confusion_matrix(y_true, y_pred,
 sample_weight=sample_weight,
 labels=labels, samplewise=samplewise)
tp_sum = MCM[:, 1, 1]
pred_sum = tp_sum + MCM[:, 0, 1]
true_sum = tp_sum + MCM[:, 1, 0]

if average == 'micro':

```

```

tp_sum = np.array([tp.sum()])
pred_sum = np.array([pred.sum()])
true_sum = np.array([true.sum()])

Finally, we have all our sufficient statistics. Divide!
beta2 = beta ** 2

Divide, and on zero-division, set scores and/or warn according to
zero_division:
precision = _prf_divide(tp_sum, pred_sum, 'precision',
 'predicted', average, warn_for, zero_division)
recall = _prf_divide(tp_sum, true_sum, 'recall',
 'true', average, warn_for, zero_division)

warn for f-score only if zero_division is warn, it is in warn_for
and BOTH prec and rec are ill-defined
if zero_division == "warn" and ("f-score",) == warn_for:
 if (pred_sum[true_sum == 0] == 0).any():
 _warn_prf(
 average, "true nor predicted", 'F-score is', len(true_sum)
)

if tp == 0 F will be 1 only if all predictions are zero, all labels are
zero, and zero_division=1. In all other case, 0
if np.isposinf(beta):
 f_score = recall
else:
 denom = beta2 * precision + recall

 denom[denom == 0.] = 1 # avoid division by 0
 f_score = (1 + beta2) * precision * recall / denom

Average the results
if average == 'weighted':
 weights = true_sum
 if weights.sum() == 0:
 zero_division_value = 0.0 if zero_division in ["warn", 0] else 1.0
 # precision is zero_division if there are no positive predictions
 # recall is zero_division if there are no positive labels
 # fscore is zero_division if all labels AND predictions are
 # negative
 return (zero_division_value if pred_sum.sum() == 0 else 0,
 zero_division_value,
 zero_division_value if pred_sum.sum() == 0 else 0,
 None)

 elif average == 'samples':
 weights = sample_weight
 else:
 weights = None

 if average is not None:
 assert average != 'binary' or len(precision) == 1
 precision = np.average(precision, weights=weights)
 recall = np.average(recall, weights=weights)
 f_score = np.average(f_score, weights=weights)
 true_sum = None # return no support

return precision, recall, f_score, true_sum

 @_deprecate_positional_args
def precision_score(y_true, y_pred, *, labels=None, pos_label=1,

```

```

 average='binary', sample_weight=None,
 zero_division="warn"):

"""Compute the precision

The precision is the ratio ``tp / (tp + fp)`` where ``tp`` is the number of
true positives and ``fp`` the number of false positives. The precision is
intuitively the ability of the classifier not to label as positive a sample
that is negative.

The best value is 1 and the worst value is 0.

Read more in the :ref:`User Guide <precision_recall_f_measure_metrics>`.

Parameters

y_true : 1d array-like, or label indicator array / sparse matrix
 Ground truth (correct) target values.

y_pred : 1d array-like, or label indicator array / sparse matrix
 Estimated targets as returned by a classifier.

labels : list, optional
 The set of labels to include when ``average != 'binary'``, and their
 order if ``average is None``. Labels present in the data can be
 excluded, for example to calculate a multiclass average ignoring a
 majority negative class, while labels not present in the data will
 result in 0 components in a macro average. For multilabel targets,
 labels are column indices. By default, all labels in ``y_true`` and
 ``y_pred`` are used in sorted order.

.. versionchanged:: 0.17
 parameter *labels* improved for multiclass problem.

pos_label : str or int, 1 by default
 The class to report if ``average='binary'`` and the data is binary.
 If the data are multiclass or multilabel, this will be ignored;
 setting ``labels=[pos_label]`` and ``average != 'binary'`` will report
 scores for that label only.

average : string, [None, 'binary' (default), 'micro', 'macro', 'samples', \
 'weighted']
 This parameter is required for multiclass/multilabel targets.
 If ``None``, the scores for each class are returned. Otherwise, this
 determines the type of averaging performed on the data:

 ``'binary'``:
 Only report results for the class specified by ``pos_label``.
 This is applicable only if targets (``y_{true,pred}``) are binary.
 ``'micro'``:
 Calculate metrics globally by counting the total true positives,
 false negatives and false positives.
 ``'macro'``:
 Calculate metrics for each label, and find their unweighted
 mean. This does not take label imbalance into account.
 ``'weighted'``:
 Calculate metrics for each label, and find their average weighted
 by support (the number of true instances for each label). This
 alters 'macro' to account for label imbalance; it can result in an
 F-score that is not between precision and recall.
 ``'samples'``:
 Calculate metrics for each instance, and find their average (only
 meaningful for multilabel classification where this differs from
 :func:`accuracy_score`).

```

```

sample_weight : array-like of shape (n_samples,), default=None
 Sample weights.

zero_division : "warn", 0 or 1, default="warn"
 Sets the value to return when there is a zero division. If set to
 "warn", this acts as 0, but warnings are also raised.

Returns

precision : float (if average is not None) or array of float, shape =\
[n_unique_labels]
 Precision of the positive class in binary classification or weighted
 average of the precision of each class for the multiclass task.

See also

precision_recall_fscore_support, multilabel_confusion_matrix

Examples

>>> from sklearn.metrics import precision_score
>>> y_true = [0, 1, 2, 0, 1, 2]
>>> y_pred = [0, 2, 1, 0, 0, 1]
>>> precision_score(y_true, y_pred, average='macro')
0.22...
>>> precision_score(y_true, y_pred, average='micro')
0.33...
>>> precision_score(y_true, y_pred, average='weighted')
0.22...
>>> precision_score(y_true, y_pred, average=None)
array([0.66..., 0. , 0.])
>>> y_pred = [0, 0, 0, 0, 0, 0]
>>> precision_score(y_true, y_pred, average=None)
array([0.33..., 0. , 0.])
>>> precision_score(y_true, y_pred, average=None, zero_division=1)
array([0.33..., 1. , 1.])

Notes

When ``true positive + false positive == 0``, precision returns 0 and
raises ``UndefinedMetricWarning``. This behavior can be
modified with ``zero_division``.

"""
p, _, _, _ = precision_recall_fscore_support(y_true, y_pred,
 labels=labels,
 pos_label=pos_label,
 average=average,
 warn_for=('precision',),
 sample_weight=sample_weight,
 zero_division=zero_division)

return p

 @_deprecate_positional_args
def recall_score(y_true, y_pred, *, labels=None, pos_label=1, average='binary',
 sample_weight=None, zero_division="warn"):
 """Compute the recall

 The recall is the ratio ``tp / (tp + fn)`` where ``tp`` is the number of
 true positives and ``fn`` the number of false negatives. The recall is
 intuitively the ability of the classifier to find all the positive samples.

```

The best value is 1 and the worst value is 0.

Read more in the :ref:`User Guide <precision\_recall\_f\_measure\_metrics>`.

**Parameters**

-----

`y_true` : 1d array-like, or label indicator array / sparse matrix  
 Ground truth (correct) target values.

`y_pred` : 1d array-like, or label indicator array / sparse matrix  
 Estimated targets as returned by a classifier.

`labels` : list, optional  
 The set of labels to include when ``average != 'binary'``, and their order if ``average is None``. Labels present in the data can be excluded, for example to calculate a multiclass average ignoring a majority negative class, while labels not present in the data will result in 0 components in a macro average. For multilabel targets, labels are column indices. By default, all labels in ``y\_true`` and ``y\_pred`` are used in sorted order.

.. versionchanged:: 0.17  
 parameter \*labels\* improved for multiclass problem.

`pos_label` : str or int, 1 by default  
 The class to report if ``average='binary'`` and the data is binary.  
 If the data are multiclass or multilabel, this will be ignored;  
 setting ``labels=[pos\_label]`` and ``average != 'binary'`` will report scores for that label only.

`average` : string, [None, 'binary' (default), 'micro', 'macro', 'samples', \  
 'weighted']  
 This parameter is required for multiclass/multilabel targets.  
 If ``None``, the scores for each class are returned. Otherwise, this determines the type of averaging performed on the data:

- ```'binary'```:  
 Only report results for the class specified by ``pos\_label``.  
 This is applicable only if targets (``y\_{true,pred}``) are binary.
- ```'micro'```:  
 Calculate metrics globally by counting the total true positives,  
 false negatives and false positives.
- ```'macro'```:  
 Calculate metrics for each label, and find their unweighted mean. This does not take label imbalance into account.
- ```'weighted'```:  
 Calculate metrics for each label, and find their average weighted by support (the number of true instances for each label). This alters 'macro' to account for label imbalance; it can result in an F-score that is not between precision and recall.
- ```'samples'```:  
 Calculate metrics for each instance, and find their average (only meaningful for multilabel classification where this differs from :func:`accuracy\_score`).

`sample_weight` : array-like of shape (n\_samples,), default=None  
 Sample weights.

`zero_division` : "warn", 0 or 1, default="warn"  
 Sets the value to return when there is a zero division. If set to "warn", this acts as 0, but warnings are also raised.

```
Returns

recall : float (if average is not None) or array of float, shape =\
 [n_unique_labels]
 Recall of the positive class in binary classification or weighted
 average of the recall of each class for the multiclass task.
```

See also

```
precision_recall_fscore_support, balanced_accuracy_score,
multilabel_confusion_matrix
```

Examples

```
>>> from sklearn.metrics import recall_score
>>> y_true = [0, 1, 2, 0, 1, 2]
>>> y_pred = [0, 2, 1, 0, 0, 1]
>>> recall_score(y_true, y_pred, average='macro')
0.33...
>>> recall_score(y_true, y_pred, average='micro')
0.33...
>>> recall_score(y_true, y_pred, average='weighted')
0.33...
>>> recall_score(y_true, y_pred, average=None)
array([1., 0., 0.])
>>> y_true = [0, 0, 0, 0, 0, 0]
>>> recall_score(y_true, y_pred, average=None)
array([0.5, 0., 0.])
>>> recall_score(y_true, y_pred, average=None, zero_division=1)
array([0.5, 1., 1.])
```

Notes

When ``true positive + false negative == 0``, recall returns 0 and raises ``UndefinedMetricWarning``. This behavior can be modified with ``zero\_division``.

```
"""
_, r, _, _ = precision_recall_fscore_support(y_true, y_pred,
 labels=labels,
 pos_label=pos_label,
 average=average,
 warn_for=('recall',),
 sample_weight=sample_weight,
 zero_division=zero_division)

return r
```

```
@_deprecate_positional_args
def balanced_accuracy_score(y_true, y_pred, *, sample_weight=None,
 adjusted=False):
 """Compute the balanced accuracy
```

The balanced accuracy in binary and multiclass classification problems to deal with imbalanced datasets. It is defined as the average of recall obtained on each class.

The best value is 1 and the worst value is 0 when ``adjusted=False``.

Read more in the :ref:`User Guide <balanced\_accuracy\_score>`.

Parameters

```

```

```
y_true : 1d array-like
```

```
Ground truth (correct) target values.

y_pred : 1d array-like
 Estimated targets as returned by a classifier.

sample_weight : array-like of shape (n_samples,), default=None
 Sample weights.

adjusted : bool, default=False
 When true, the result is adjusted for chance, so that random
 performance would score 0, and perfect performance scores 1.

Returns

balanced_accuracy : float

See also

recall_score, roc_auc_score

Notes

Some literature promotes alternative definitions of balanced accuracy. Our
definition is equivalent to :func:`accuracy_score` with class-balanced
sample weights, and shares desirable properties with the binary case.
See the :ref:`User Guide <balanced_accuracy_score>`.

References

.. [1] Brodersen, K.H.; Ong, C.S.; Stephan, K.E.; Buhmann, J.M. (2010).
 The balanced accuracy and its posterior distribution.
 Proceedings of the 20th International Conference on Pattern
 Recognition, 3121-24.
.. [2] John. D. Kelleher, Brian Mac Namee, Aoife D'Arcy, (2015).
 `Fundamentals of Machine Learning for Predictive Data Analytics:
 Algorithms, Worked Examples, and Case Studies
 <https://mitpress.mit.edu/books/fundamentals-machine-learning-predictive-data-analytics>`_.

Examples

>>> from sklearn.metrics import balanced_accuracy_score
>>> y_true = [0, 1, 0, 0, 1, 0]
>>> y_pred = [0, 1, 0, 0, 0, 1]
>>> balanced_accuracy_score(y_true, y_pred)
0.625

"""
C = confusion_matrix(y_true, y_pred, sample_weight=sample_weight)
with np.errstate(divide='ignore', invalid='ignore'):
 per_class = np.diag(C) / C.sum(axis=1)
if np.any(np.isnan(per_class)):
 warnings.warn('y_pred contains classes not in y_true')
 per_class = per_class[~np.isnan(per_class)]
score = np.mean(per_class)
if adjusted:
 n_classes = len(per_class)
 chance = 1 / n_classes
 score -= chance
 score /= 1 - chance
return score
```

```
@_deprecate_positional_args
def classification_report(y_true, y_pred, *, labels=None, target_names=None,
 sample_weight=None, digits=2, output_dict=False,
 zero_division="warn"):
 """Build a text report showing the main classification metrics.
```

Read more in the :ref:`User Guide <classification\_report>`.

#### Parameters

-----

y\_true : 1d array-like, or label indicator array / sparse matrix  
Ground truth (correct) target values.

y\_pred : 1d array-like, or label indicator array / sparse matrix  
Estimated targets as returned by a classifier.

labels : array, shape = [n\_labels]  
Optional list of label indices to include in the report.

target\_names : list of strings  
Optional display names matching the labels (same order).

sample\_weight : array-like of shape (n\_samples,), default=None  
Sample weights.

digits : int  
Number of digits for formatting output floating point values.  
When ``output\_dict`` is ``True``, this will be ignored and the  
returned values will not be rounded.

output\_dict : bool (default = False)  
If True, return output as dict

zero\_division : "warn", 0 or 1, default="warn"  
Sets the value to return when there is a zero division. If set to  
"warn", this acts as 0, but warnings are also raised.

#### Returns

-----

report : string / dict  
Text summary of the precision, recall, F1 score for each class.  
Dictionary returned if output\_dict is True. Dictionary has the  
following structure:::

```
{'label 1': {'precision':0.5,
 'recall':1.0,
 'f1-score':0.67,
 'support':1},
 'label 2': { ... },
 ...
}
```

The reported averages include macro average (averaging the unweighted mean per label), weighted average (averaging the support-weighted mean per label), and sample average (only for multilabel classification). Micro average (averaging the total true positives, false negatives and false positives) is only shown for multi-label or multi-class with a subset of classes, because it corresponds to accuracy otherwise. See also :func:`precision\_recall\_fscore\_support` for more details on averages.

Note that in binary classification, recall of the positive class is also known as "sensitivity"; recall of the negative class is

```
"specificity".
```

See also

-----  
`precision_recall_fscore_support`, `confusion_matrix`,  
`multilabel_confusion_matrix`

Examples

```

>>> from sklearn.metrics import classification_report
>>> y_true = [0, 1, 2, 2, 2]
>>> y_pred = [0, 0, 2, 2, 1]
>>> target_names = ['class 0', 'class 1', 'class 2']
>>> print(classification_report(y_true, y_pred, target_names=target_names))
 precision recall f1-score support
<BLANKLINE>
 class 0 0.50 1.00 0.67 1
 class 1 0.00 0.00 0.00 1
 class 2 1.00 0.67 0.80 3
<BLANKLINE>
 accuracy 0.60 5
 macro avg 0.50 0.56 0.49 5
weighted avg 0.70 0.60 0.61 5
<BLANKLINE>
>>> y_pred = [1, 1, 0]
>>> y_true = [1, 1, 1]
>>> print(classification_report(y_true, y_pred, labels=[1, 2, 3]))
 precision recall f1-score support
<BLANKLINE>
 1 1.00 0.67 0.80 3
 2 0.00 0.00 0.00 0
 3 0.00 0.00 0.00 0
<BLANKLINE>
 micro avg 1.00 0.67 0.80 3
 macro avg 0.33 0.22 0.27 3
weighted avg 1.00 0.67 0.80 3
<BLANKLINE>
"""

y_type, y_true, y_pred = _check_targets(y_true, y_pred)

labels_given = True
if labels is None:
 labels = unique_labels(y_true, y_pred)
 labels_given = False
else:
 labels = np.asarray(labels)

labelled micro average
micro_is_accuracy = ((y_type == 'multiclass' or y_type == 'binary') and
 (not labels_given or
 (set(labels) == set(unique_labels(y_true, y_pred)))))

if target_names is not None and len(labels) != len(target_names):
 if labels_given:
 warnings.warn(
 "labels size, {0}, does not match size of target_names, {1}"
 .format(len(labels), len(target_names))
)
 else:
 raise ValueError(
 "Number of classes, {0}, does not match size of "
 "target_names, {1}. Try specifying the labels "
```

```

 "parameter".format(len(labels), len(target_names))
)
if target_names is None:
 target_names = ['%s' % l for l in labels]

headers = ["precision", "recall", "f1-score", "support"]
compute per-class results without averaging
p, r, f1, s = precision_recall_fscore_support(y_true, y_pred,
 labels=labels,
 average=None,
 sample_weight=sample_weight,
 zero_division=zero_division)
rows = zip(target_names, p, r, f1, s)

if y_type.startswith('multilabel'):
 average_options = ('micro', 'macro', 'weighted', 'samples')
else:
 average_options = ('micro', 'macro', 'weighted')

if output_dict:
 report_dict = {label[0]: label[1:] for label in rows}
 for label, scores in report_dict.items():
 report_dict[label] = dict(zip(headers,
 [i.item() for i in scores]))
else:
 longest_last_line_heading = 'weighted avg'
 name_width = max(len(cn) for cn in target_names)
 width = max(name_width, len(longest_last_line_heading), digits)
 head_fmt = '{:>{width}s} ' + '{:>9}' * len(headers)
 report = head_fmt.format('', *headers, width=width)
 report += '\n\n'
 row_fmt = '{:>{width}s} ' + '{:>9.{digits}f}' * 3 + '{:>9}\n'
 for row in rows:
 report += row_fmt.format(*row, width=width, digits=digits)
 report += '\n'

compute all applicable averages
for average in average_options:
 if average.startswith('micro') and micro_is_accuracy:
 line_heading = 'accuracy'
 else:
 line_heading = average + ' avg'

 # compute averages with specified averaging method
 avg_p, avg_r, avg_f1, _ = precision_recall_fscore_support(
 y_true, y_pred, labels=labels,
 average=average, sample_weight=sample_weight,
 zero_division=zero_division)
 avg = [avg_p, avg_r, avg_f1, np.sum(s)]

 if output_dict:
 report_dict[line_heading] = dict(
 zip(headers, [i.item() for i in avg]))
 else:
 if line_heading == 'accuracy':
 row_fmt_accuracy = '{:>{width}s} ' + \
 '{:>9.{digits}}' * 2 + '{:>9.{digits}f}' + \
 '{:>9}\n'
 report += row_fmt_accuracy.format(line_heading, '',
 *avg[2:], width=width,
 digits=digits)
 else:
 report += row_fmt.format(line_heading, *avg,

```

```
width=width, digits=digits)

if output_dict:
 if 'accuracy' in report_dict.keys():
 report_dict['accuracy'] = report_dict['accuracy']['precision']
 return report_dict
else:
 return report

 @_deprecate_positional_args
def hamming_loss(y_true, y_pred, *, sample_weight=None):
 """Compute the average Hamming loss.

 The Hamming loss is the fraction of labels that are incorrectly predicted.

 Read more in the :ref:`User Guide <hamming_loss>`.

 Parameters

 y_true : 1d array-like, or label indicator array / sparse matrix
 Ground truth (correct) labels.

 y_pred : 1d array-like, or label indicator array / sparse matrix
 Predicted labels, as returned by a classifier.

 sample_weight : array-like of shape (n_samples,), default=None
 Sample weights.

 .. versionadded:: 0.18

 Returns

 loss : float or int,
 Return the average Hamming loss between element of ``y_true`` and
 ``y_pred``.

 See Also

 accuracy_score, jaccard_score, zero_one_loss

 Notes

 In multiclass classification, the Hamming loss corresponds to the Hamming
 distance between ``y_true`` and ``y_pred`` which is equivalent to the
 subset ``zero_one_loss`` function, when `normalize` parameter is set to
 True.

 In multilabel classification, the Hamming loss is different from the
 subset zero-one loss. The zero-one loss considers the entire set of labels
 for a given sample incorrect if it does not entirely match the true set of
 labels. Hamming loss is more forgiving in that it penalizes only the
 individual labels.

 The Hamming loss is upperbounded by the subset zero-one loss, when
 `normalize` parameter is set to True. It is always between 0 and 1,
 lower being better.

 References

 .. [1] Grigorios Tsoumakas, Ioannis Katakis. Multi-Label Classification:
 An Overview. International Journal of Data Warehousing & Mining,
 3(3), 1-13, July-September 2007.
```

```
.. [2] `Wikipedia entry on the Hamming distance
 <https://en.wikipedia.org/wiki/Hamming_distance>`_
```

## Examples

-----

```
>>> from sklearn.metrics import hamming_loss
>>> y_pred = [1, 2, 3, 4]
>>> y_true = [2, 2, 3, 4]
>>> hamming_loss(y_true, y_pred)
0.25
```

In the multilabel case with binary label indicators:

```
>>> import numpy as np
>>> hamming_loss(np.array([[0, 1], [1, 1]]), np.zeros((2, 2)))
0.75
"""
y_type, y_true, y_pred = _check_targets(y_true, y_pred)
check_consistent_length(y_true, y_pred, sample_weight)

if sample_weight is None:
 weight_average = 1.
else:
 weight_average = np.mean(sample_weight)

if y_type.startswith('multilabel'):
 n_differences = count_nonzero(y_true - y_pred,
 sample_weight=sample_weight)
 return (n_differences /
 (y_true.shape[0] * y_true.shape[1] * weight_average))

elif y_type in ["binary", "multiclass"]:
 return _weighted_sum(y_true != y_pred, sample_weight, normalize=True)
else:
 raise ValueError("{0} is not supported".format(y_type))

@_deprecate_positional_args
def log_loss(y_true, y_pred, *, eps=1e-15, normalize=True, sample_weight=None,
 labels=None):
 """Log loss, aka logistic loss or cross-entropy loss.
```

This is the loss function used in (multinomial) logistic regression and extensions of it such as neural networks, defined as the negative log-likelihood of a logistic model that returns ``y\_pred`` probabilities for its training data ``y\_true``.

The log loss is only defined for two or more labels.

For a single sample with true label  $y_t$  in  $\{0, 1\}$  and estimated probability  $y_p$  that  $y_t = 1$ , the log loss is

$$-\log P(y_t|y_p) = -(y_t \log(y_p) + (1 - y_t) \log(1 - y_p))$$

Read more in the :ref:`User Guide <log\_loss>`.

## Parameters

-----

`y_true` : array-like or label indicator matrix  
Ground truth (correct) labels for `n_samples` samples.

`y_pred` : array-like of float, shape = (`n_samples`, `n_classes`) or (`n_samples`,)  
Predicted probabilities, as returned by a classifier's



```

transformed_labels = lb.transform(y_true)

if transformed_labels.shape[1] == 1:
 transformed_labels = np.append(1 - transformed_labels,
 transformed_labels, axis=1)

Clipping
y_pred = np.clip(y_pred, eps, 1 - eps)

If y_pred is of single dimension, assume y_true to be binary
and then check.
if y_pred.ndim == 1:
 y_pred = y_pred[:, np.newaxis]
if y_pred.shape[1] == 1:
 y_pred = np.append(1 - y_pred, y_pred, axis=1)

Check if dimensions are consistent.
transformed_labels = check_array(transformed_labels)
if len(lb.classes_) != y_pred.shape[1]:
 if labels is None:
 raise ValueError("y_true and y_pred contain different number of "
 "classes {0}, {1}. Please provide the true "
 "labels explicitly through the labels argument. "
 "Classes found in "
 "y_true: {2}".format(transformed_labels.shape[1],
 y_pred.shape[1],
 lb.classes_))
 else:
 raise ValueError('The number of classes in labels is different '
 'from that in y_pred. Classes found in '
 'labels: {0}'.format(lb.classes_))

Renormalize
y_pred /= y_pred.sum(axis=1)[:, np.newaxis]
loss = -(transformed_labels * np.log(y_pred)).sum(axis=1)

return _weighted_sum(loss, sample_weight, normalize)

```

`@_deprecate_positional_args`  
`def hinge_loss(y_true, pred_decision, *, labels=None, sample_weight=None):`  
 `"""Average hinge loss (non-regularized)`

In binary class case, assuming labels in `y_true` are encoded with +1 and -1, when a prediction mistake is made, ``margin = y_true * pred_decision`` is always negative (since the signs disagree), implying ``1 - margin`` is always greater than 1. The cumulated hinge loss is therefore an upper bound of the number of mistakes made by the classifier.

In multiclass case, the function expects that either all the labels are included in `y_true` or an optional `labels` argument is provided which contains all the labels. The multilabel margin is calculated according to Crammer-Singer's method. As in the binary case, the cumulated hinge loss is an upper bound of the number of mistakes made by the classifier.

Read more in the :ref:`User Guide <hinge\_loss>`.

#### Parameters

-----

`y_true` : array, shape = [n\_samples]  
 True target, consisting of integers of two values. The positive label must be greater than the negative label.

```
pred_decision : array, shape = [n_samples] or [n_samples, n_classes]
 Predicted decisions, as output by decision_function (floats).

labels : array, optional, default None
 Contains all the labels for the problem. Used in multiclass hinge loss.

sample_weight : array-like of shape (n_samples,), default=None
 Sample weights.

Returns

loss : float

References

.. [1] `Wikipedia entry on the Hinge loss
 <https://en.wikipedia.org/wiki/Hinge_loss>`_

.. [2] Koby Crammer, Yoram Singer. On the Algorithmic
 Implementation of Multiclass Kernel-based Vector
 Machines. Journal of Machine Learning Research 2,
 (2001), 265-292

.. [3] `L1 AND L2 Regularization for Multiclass Hinge Loss Models
 by Robert C. Moore, John DeNero.
 <http://www.ttic.edu/sigml/symposium2011/papers/
 Moore+DeNero-Regularization.pdf>`_
```

## Examples

```
>>> from sklearn import svm
>>> from sklearn.metrics import hinge_loss
>>> X = [[0], [1]]
>>> y = [-1, 1]
>>> est = svm.LinearSVC(random_state=0)
>>> est.fit(X, y)
LinearSVC(random_state=0)
>>> pred_decision = est.decision_function([-2, 3, 0.5])
>>> pred_decision
array([-2.18..., 2.36..., 0.09...])
>>> hinge_loss([-1, 1, 1], pred_decision)
0.30...
```

In the multiclass case:

```
>>> import numpy as np
>>> X = np.array([[0], [1], [2], [3]])
>>> Y = np.array([0, 1, 2, 3])
>>> labels = np.array([0, 1, 2, 3])
>>> est = svm.LinearSVC()
>>> est.fit(X, Y)
LinearSVC()
>>> pred_decision = est.decision_function([-1, 2, 3])
>>> y_true = [0, 2, 3]
>>> hinge_loss(y_true, pred_decision, labels=labels)
0.56...
"""
check_consistent_length(y_true, pred_decision, sample_weight)
pred_decision = check_array(pred_decision, ensure_2d=False)
y_true = column_or_1d(y_true)
y_true_unique = np.unique(y_true)
if y_true_unique.size > 2:
 if (labels is None and pred_decision.ndim > 1 and
```

```

 (np.size(y_true_unique) != pred_decision.shape[1])):
 raise ValueError("Please include all labels in y_true "
 "or pass labels as third argument")
if labels is None:
 labels = y_true_unique
le = LabelEncoder()
le.fit(labels)
y_true = le.transform(y_true)
mask = np.ones_like(pred_decision, dtype=bool)
mask[np.arange(y_true.shape[0]), y_true] = False
margin = pred_decision[~mask]
margin -= np.max(pred_decision[mask].reshape(y_true.shape[0], -1),
 axis=1)

else:
 # Handles binary class case
 # this code assumes that positive and negative labels
 # are encoded as +1 and -1 respectively
 pred_decision = column_or_1d(pred_decision)
 pred_decision = np.ravel(pred_decision)

 lbin = LabelBinarizer(neg_label=-1)
 y_true = lbin.fit_transform(y_true)[:, 0]

 try:
 margin = y_true * pred_decision
 except TypeError:
 raise TypeError("pred_decision should be an array of floats.")

losses = 1 - margin
The hinge_loss doesn't penalize good enough predictions.
np.clip(losses, 0, None, out=losses)
return np.average(losses, weights=sample_weight)

@_deprecate_positional_args
def brier_score_loss(y_true, y_prob, *, sample_weight=None, pos_label=None):
 """Compute the Brier score.

The smaller the Brier score, the better, hence the naming with "loss". Across all items in a set N predictions, the Brier score measures the mean squared difference between (1) the predicted probability assigned to the possible outcomes for item i, and (2) the actual outcome. Therefore, the lower the Brier score is for a set of predictions, the better the predictions are calibrated. Note that the Brier score always takes on a value between zero and one, since this is the largest possible difference between a predicted probability (which must be between zero and one) and the actual outcome (which can take on values of only 0 and 1). The Brier loss is composed of refinement loss and calibration loss. The Brier score is appropriate for binary and categorical outcomes that can be structured as true or false, but is inappropriate for ordinal variables which can take on three or more values (this is because the Brier score assumes that all possible outcomes are equivalently "distant" from one another). Which label is considered to be the positive label is controlled via the parameter pos_label, which defaults to 1. Read more in the :ref:`User Guide <calibration>`.
```

#### Parameters

-----

y\_true : array, shape (n\_samples,)
True targets.

```
y_prob : array, shape (n_samples,)
 Probabilities of the positive class.

sample_weight : array-like of shape (n_samples,), default=None
 Sample weights.

pos_label : int or str, default=None
 Label of the positive class.
 Defaults to the greater label unless y_true is all 0 or all -1
 in which case pos_label defaults to 1.
```

Returns

-----

```
score : float
 Brier score
```

Examples

-----

```
>>> import numpy as np
>>> from sklearn.metrics import brier_score_loss
>>> y_true = np.array([0, 1, 1, 0])
>>> y_true_categorical = np.array(["spam", "ham", "ham", "spam"])
>>> y_prob = np.array([0.1, 0.9, 0.8, 0.3])
>>> brier_score_loss(y_true, y_prob)
0.037...
>>> brier_score_loss(y_true, 1-y_prob, pos_label=0)
0.037...
>>> brier_score_loss(y_true_categorical, y_prob, pos_label="ham")
0.037...
>>> brier_score_loss(y_true, np.array(y_prob) > 0.5)
0.0
```

References

-----

```
.. [1] `Wikipedia entry for the Brier score.
 <https://en.wikipedia.org/wiki/Brier_score>`_
"""

y_true = column_or_1d(y_true)
y_prob = column_or_1d(y_prob)
assert_all_finite(y_true)
assert_all_finite(y_prob)
check_consistent_length(y_true, y_prob, sample_weight)

labels = np.unique(y_true)
if len(labels) > 2:
 raise ValueError("Only binary classification is supported. "
 "Labels in y_true: %s." % labels)
if y_prob.max() > 1:
 raise ValueError("y_prob contains values greater than 1.")
if y_prob.min() < 0:
 raise ValueError("y_prob contains values less than 0.")

if pos_label=None, when y_true is in {-1, 1} or {0, 1},
pos_label is set to 1 (consistent with precision_recall_curve/roc_curve),
otherwise pos_label is set to the greater label
(different from precision_recall_curve/roc_curve,
the purpose is to keep backward compatibility).
if pos_label is None:
 if (np.array_equal(labels, [0]) or
 np.array_equal(labels, [-1])):
 pos_label = 1
 else:
 pos_label = y_true.max()
```

```
y_true = np.array(y_true == pos_label, int)
return np.average((y_true - y_prob) ** 2, weights=sample_weight)
```

## Tutorial

1. Download Python - <https://www.python.org/downloads/>

Note: *at the time of writing, version 3.7 was used – latest versions should work as well*

2. Install Jupyter Notebook using pip - <https://jupyter.org/install>

Note: *version info at the time of writing – latest versions should work as well*

```
jupyter core: 4.6.3
jupyter-notebook: 6.0.3
qtconsole: 4.7.2
ipython: 7.13.0
ipykernel: 5.2.0
jupyter client: 6.1.2
jupyter lab: not installed
nbconvert: 5.6.1
ipywidgets: 7.5.1
nbformat: 5.0.5
traitlets: 4.3.3
```

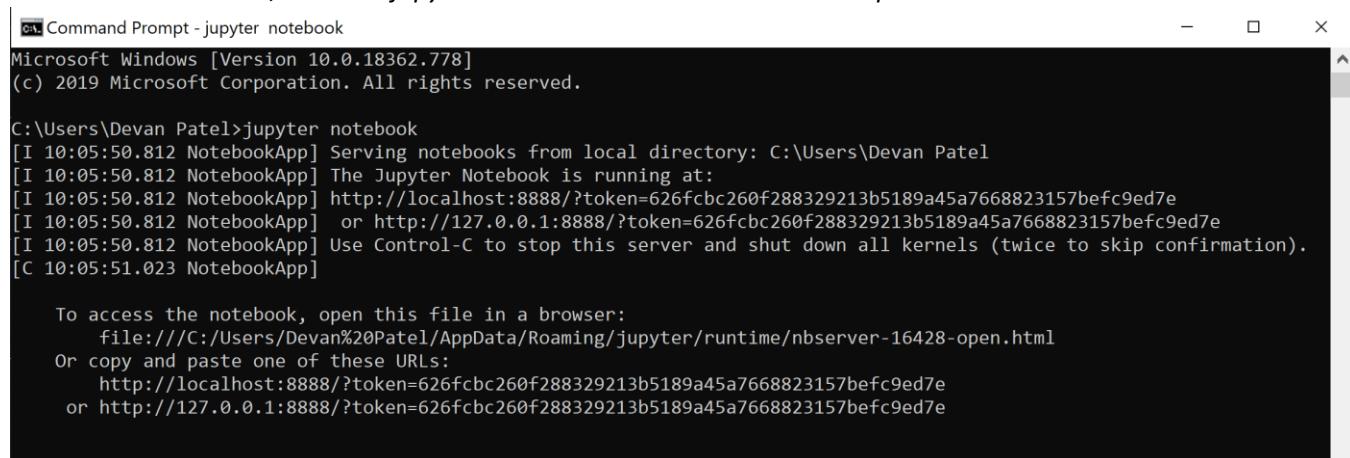
3. Install *pandas v1.0.3, numpy v1.18.2, matplotlib v3.2.1, seaborn v0.10.0, sklearn v0.0* using pip (e.g. `pip install sklearn`)

Note: *you may install the latest versions – version info is provided in case newer versions do not provide backwards compatibility*

4. Download dataset from <https://www.kaggle.com/uciml/mushroom-classification>

5. Start Jupyter Notebook

Note: *On Windows 10, execute ‘jupyter notebook’ in the Command Prompt.*

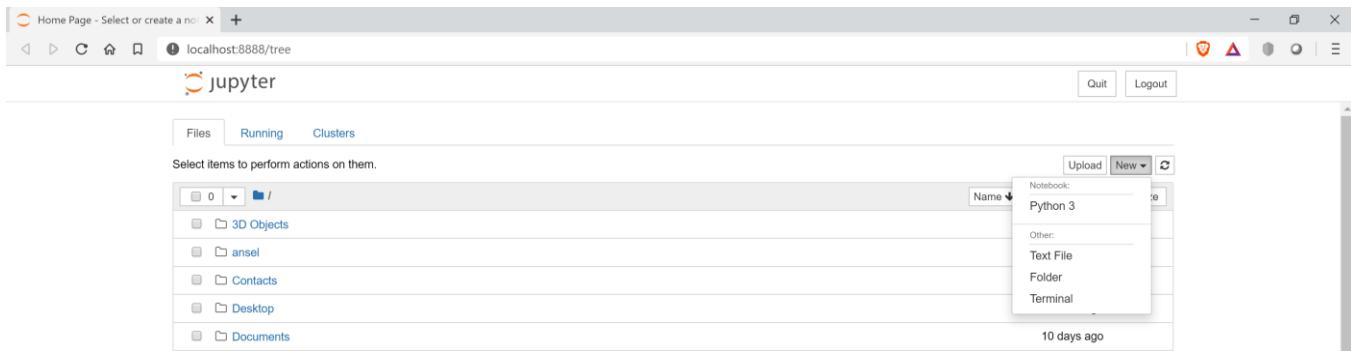


```
Command Prompt - jupyter notebook
Microsoft Windows [Version 10.0.18362.778]
(c) 2019 Microsoft Corporation. All rights reserved.

C:\Users\Devan Patel>jupyter notebook
[I 10:05:50.812 NotebookApp] Serving notebooks from local directory: C:\Users\Devan Patel
[I 10:05:50.812 NotebookApp] The Jupyter Notebook is running at:
[I 10:05:50.812 NotebookApp] http://localhost:8888/?token=626fcbe260f288329213b5189a45a7668823157befc9ed7e
[I 10:05:50.812 NotebookApp] or http://127.0.0.1:8888/?token=626fcbe260f288329213b5189a45a7668823157befc9ed7e
[I 10:05:50.812 NotebookApp] Use Control-C to stop this server and shut down all kernels (twice to skip confirmation).
[C 10:05:51.023 NotebookApp]

To access the notebook, open this file in a browser:
 file:///C:/Users/Devan%20Patel/AppData/Roaming/jupyter/runtime/nbserver-16428-open.html
Or copy and paste one of these URLs:
 http://localhost:8888/?token=626fcbe260f288329213b5189a45a7668823157befc9ed7e
 or http://127.0.0.1:8888/?token=626fcbe260f288329213b5189a45a7668823157befc9ed7e
```

6. Go to <http://localhost:8888/tree>, click ‘New’ and then click ‘Python 3’ to create a notebook for this project.



## 7. Import required libraries

```
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
import seaborn as sns
from sklearn.preprocessing import LabelEncoder
from sklearn.model_selection import train_test_split
from sklearn.ensemble import RandomForestClassifier
from sklearn.naive_bayes import CategoricalNB
from sklearn.model_selection import KFold
from sklearn.metrics import confusion_matrix, accuracy_score, precision_score, recall_score, f1_score
```

## 8. Read the csv file into `pandas.DataFrame` and visualize the data. Replace `path` with either the relative or absolute path to the csv file downloaded from Kaggle.

```
data = pd.read_csv(path)
data.head()
```

	class	cap-shape	cap-surface	cap-color	bruises	odor	gill-attachment	gill-spacing	gill-size	gill-color	...	stalk-surface-below-ring	stalk-color-above-ring	stalk-below-ring	veil-type	veil-color	ring-number	ring-type	spore-print-color	populat
0	p	x	s	n	t	p	f	c	n	k	...	s	w	w	p	w	o	p	k	
1	e	x	s	y	t	a	f	c	b	k	...	s	w	w	p	w	o	p	n	
2	e	b	s	w	t	l	f	c	b	n	...	s	w	w	p	w	o	p	n	
3	p	x	y	w	t	p	f	c	n	n	...	s	w	w	p	w	o	p	k	
4	e	x	s	g	f	n	f	w	b	k	...	s	w	w	p	w	o	e	n	

5 rows × 23 columns

Category names have been shortened in the dataset to one letter and the mapping can be found on Kaggle. Please take a careful look at this mapping as it is important to understand the data before processing it.

## 9. Let us get a clear view on what features are available in our dataset by calling the `info` method on the DataFrame.

```
data.info()
```

```

<class 'pandas.core.frame.DataFrame'>
RangeIndex: 8124 entries, 0 to 8123
Data columns (total 23 columns):
 # Column Non-Null Count Dtype

 0 class 8124 non-null object
 1 cap-shape 8124 non-null object
 2 cap-surface 8124 non-null object
 3 cap-color 8124 non-null object
 4 bruises 8124 non-null object
 5 odor 8124 non-null object
 6 gill-attachment 8124 non-null object
 7 gill-spacing 8124 non-null object
 8 gill-size 8124 non-null object
 9 gill-color 8124 non-null object
 10 stalk-shape 8124 non-null object
 11 stalk-root 8124 non-null object
 12 stalk-surface-above-ring 8124 non-null object
 13 stalk-surface-below-ring 8124 non-null object
 14 stalk-color-above-ring 8124 non-null object
 15 stalk-color-below-ring 8124 non-null object
 16 veil-type 8124 non-null object
 17 veil-color 8124 non-null object
 18 ring-number 8124 non-null object
 19 ring-type 8124 non-null object
 20 spore-print-color 8124 non-null object
 21 population 8124 non-null object
 22 habitat 8124 non-null object
dtypes: object(23)
memory usage: 1.4+ MB

```

As we can see, there are a total of 8124 observations, 23 features and none of the features have a single null value.

10. We can confirm that there are no nulls in our dataset by observing the output of the following code:

```

data.isnull().sum()
class 0
cap-shape 0
cap-surface 0
cap-color 0
bruises 0
odor 0
gill-attachment 0
gill-spacing 0
gill-size 0
gill-color 0
stalk-shape 0
stalk-root 0
stalk-surface-above-ring 0
stalk-surface-below-ring 0
stalk-color-above-ring 0
stalk-color-below-ring 0
veil-type 0
veil-color 0
ring-number 0
ring-type 0
spore-print-color 0
population 0
habitat 0
dtype: int64

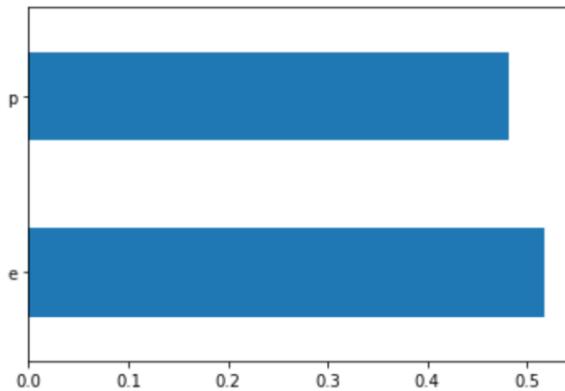
```

11. We will be predicting whether a mushroom is poisonous or edible. Let us make sure our dataset is balanced. By looking at the bar graph, we observe that there are nearly as many observations in the dataset for poisonous mushrooms as there are for edible mushrooms and hence, our dataset is balanced.

```

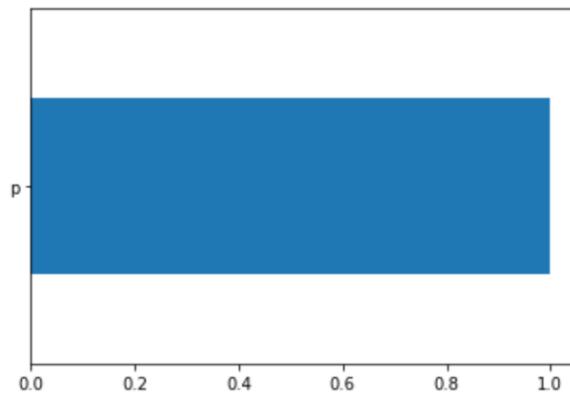
data['class'].value_counts(normalize=True).barplot(kind='barh')

```



12. I checked the frequencies of all the features and found that all observations fall into the same class of veil-type. This means that the feature adds no value to our models since it will have no role in the classification. We can drop it to speed up training.

```
data['veil-type'].value_counts(normalize=True).plot(kind='barh')
```



13. We can drop features using the drop method. A list of feature names is provided as the first parameter.

```
data.drop(['veil-type'], axis=1, inplace=True)
```

14. All features in the dataset are categorical and hence, we must encode them numerically. For example, there are two possible values for the class feature – p or e and hence, we must make it so that p corresponds to 0 and e corresponds to 1 (or vice-versa). If we do not encode our features to be numeric, we will have runtime errors. We can encode all of our features by iterating through all the column names and transforming the data using [LabelEncoder](#) from `sklearn.preprocessing`.

```
le = LabelEncoder()
for c in data.columns:
 data[c] = le.fit_transform(data[c])
```

15. We will now prepare a DataFrame that can be used to [draw a violin plot](#). The following code will pivot our data at class (feature that we are trying to predict) and provide the value of all other features as separate rows.

```
df_div = pd.melt(data, 'class', var_name = 'features')
```

16. Let us look at the newly created DataFrame to understand what we did.

```
df_div.head()
```

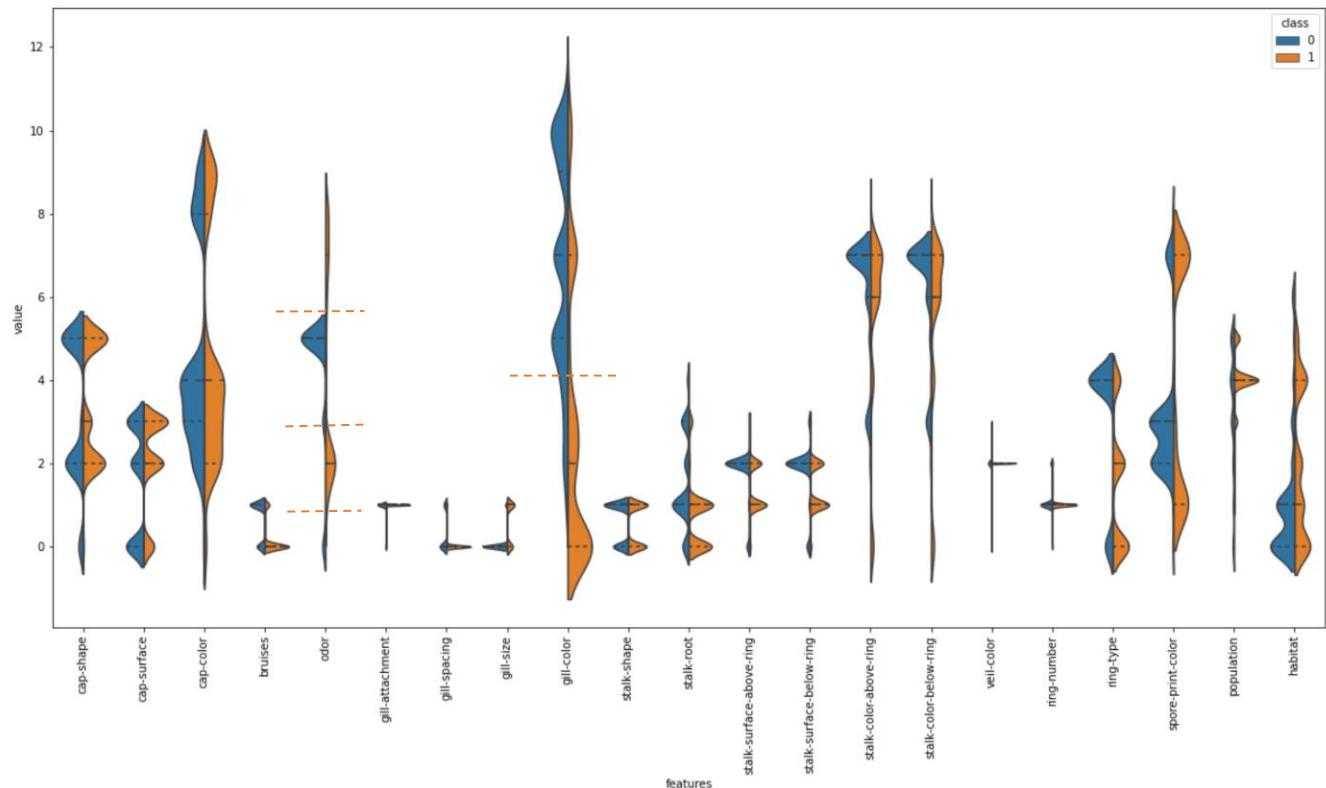
	class	features	value
0	1	cap-shape	5
1	0	cap-shape	5
2	0	cap-shape	0
3	1	cap-shape	5
4	0	cap-shape	5

```
df_div.tail()
```

	class	features	value
170599	0	habitat	2
170600	0	habitat	2
170601	0	habitat	2
170602	1	habitat	2
170603	0	habitat	2

17. A typical violin plot provides the same information a box-and-whisker plot does, combined with information about density. You can set the parameter inner of the violin plot method to 'box' instead of 'quartile' to draw a box-and-whisker plot at the center of each violin. You may have also noticed that the parameter split is set to True. You may try different inputs to the parameter and see the results yourself. The parameter split is set to True so that we can easily compare the distribution of poisonous and edible mushrooms for each feature. Particularly, we will use this graph to get a general intuition regarding which features are important.

```
fig, ax = plt.subplots(figsize = (20, 10))
p = sns.violinplot(
 ax = ax,
 data = df_div,
 x = 'features',
 y = 'value',
 hue = "class",
 split = True,
 inner = 'quartile',
)
df_no_class = data.drop(['class'], axis = 1)
p.set_xticklabels(rotation = 90, labels = list(df_no_class.columns));
```



Notice that gill-color is quite an important decision maker. At a first glance, we can see that the bottom third of the violin for gill-color mostly falls into class 1 and the rest mostly falls into class 0. The odor of the mushroom is

also important, and we can observe this by cutting the violin graph for odor at several places. I have placed orange dashed lines wherever we should observe the cuts.

18. We do not want to train our models with duplicate observations since doing so would introduce bias to our models. There were no duplicates in this dataset at the time of use. You can double check using the following code.

```
data.duplicated().value_counts()
False 8124
dtype: int64
```

19. We will be using k-folds cross validation as a technique to detect overfitting. A clear sign of overfitting is when the model is good at predicting what it has seen (data that has been used to train it) but not good at predicting data that it has not seen. Basically, the model has learnt to memorize answers but did not learn the patterns in the data.

```
y = data['class']
X = data.loc[:, data.columns != 'class']
kf = KFold(n_splits=10)
random_forest_classifier_accuracy_scores = []
random_forest_classifier_precision_scores = []
random_forest_classifier_recall_scores = []
random_forest_classifier_f1_scores = []
naive_bayes_classifier_accuracy_scores = []
naive_bayes_classifier_precision_scores = []
naive_bayes_classifier_recall_scores = []
naive_bayes_classifier_f1_scores = []
random_forest_classifier_confusion_matrices = []
naive_bayes_classifier_confusion_matrices = []
for train_index, test_index in kf.split(X):
 X_train, X_test = X.iloc[train_index], X.iloc[test_index]
 y_train, y_test = y.iloc[train_index], y.iloc[test_index]
 random_forest_classifier = RandomForestClassifier()
 random_forest_classifier.fit(X_train, y_train)
 naive_bayes_classifier = CategoricalNB()
 naive_bayes_classifier.fit(X_train, y_train)

 random_forest_classifier_y_pred = random_forest_classifier.predict(X_test)
 random_forest_classifier_accuracy_scores.append(accuracy_score(y_test, random_forest_classifier_y_pred))
 random_forest_classifier_precision_scores.append(precision_score(y_test, random_forest_classifier_y_pred))
 random_forest_classifier_recall_scores.append(recall_score(y_test, random_forest_classifier_y_pred))
 random_forest_classifier_f1_scores.append(f1_score(y_test, random_forest_classifier_y_pred))
 random_forest_classifier_confusion_matrices.append(confusion_matrix(y_test, random_forest_classifier_y_pred).ravel())
 naive_bayes_classifier_y_pred = naive_bayes_classifier.predict(X_test)
 naive_bayes_classifier_accuracy_scores.append(accuracy_score(y_test, naive_bayes_classifier_y_pred))
 naive_bayes_classifier_precision_scores.append(precision_score(y_test, naive_bayes_classifier_y_pred))
```

```

 naive_bayes_classifier_recall_scores.append(recall_score(y_test, naive_bayes_classifier_y_pred))
 naive_bayes_classifier_f1_scores.append(f1_score(y_test, naive_bayes_classifier_y_pred))
 naive_bayes_classifier_confusion_matrices.append(confusion_matrix(y_test, naive_bayes_classifier_y_pred).ravel())

print('--- Random Forest Classifier---')
print('Average Accuracy:',\
 sum(random_forest_classifier_accuracy_scores)/len(random_forest_classifier_accuracy_scores))
print('Average Precision:',\
 sum(random_forest_classifier_precision_scores)/len(random_forest_classifier_precision_scores))
print('Average Recall:',\
 sum(random_forest_classifier_recall_scores)/len(random_forest_classifier_recall_scores))
print('Average F1:',\
 sum(random_forest_classifier_f1_scores)/len(random_forest_classifier_f1_scores))
print("TN, FP, FN, TP")
for i in random_forest_classifier_confusion_matrices:
 print(i)

print('--- Naive Bayes Classifier ---')
print('Average Accuracy:',\
 sum(naive_bayes_classifier_accuracy_scores)/len(naive_bayes_classifier_accuracy_scores))
print('Average Precision:',\
 sum(naive_bayes_classifier_precision_scores)/len(naive_bayes_classifier_accuracy_scores))
print('Average Recall:',\
 sum(naive_bayes_classifier_recall_scores)/len(naive_bayes_classifier_recall_scores))
print('Average F1:',\
 sum(naive_bayes_classifier_f1_scores)/len(naive_bayes_classifier_f1_scores))
print("TN, FP, FN, TP")
for i in naive_bayes_classifier_confusion_matrices:
 print(i)

```

```

--- Random Forest Classifier---
Average Accuracy: 1.0
Average Precision: 1.0
Average Recall: 1.0
Average F1: 1.0
TN, FP, FN, TP
[729 0 0 84]
[703 0 0 110]
[745 0 0 68]
[697 0 0 116]
[453 0 0 359]
[98 0 0 714]
[175 0 0 637]
[101 0 0 711]
[148 0 0 664]
[359 0 0 453]
--- Naive Bayes Classifier ---
Average Accuracy: 0.9404299892752622
Average Precision: 0.9908417131287326
Average Recall: 0.6955329905439023
Average F1: 0.751261439894052
TN, FP, FN, TP
[729 0 50 34]
[703 0 105 5]
[745 0 58 10]
[697 0 24 92]
[453 0 109 250]
[96 2 9 705]
[153 22 46 591]
[99 2 30 681]
[143 5 0 664]
[339 20 2 451]

```

20. Let us take the last model built using [RandomForestClassifier](#) to get information about importance of the features.

```

features_list = X.columns.values
feature_importance = random_forest_classifier.feature_importances_
sorted_idx = np.argsort(feature_importance)
plt.figure(figsize=(5,7))
plt.barh(range(len(sorted_idx)), feature_importance[sorted_idx], align='center')
plt.yticks(range(len(sorted_idx)), features_list[sorted_idx])
plt.xlabel('Importance')
plt.title('Feature Importances')
plt.draw()
plt.show()

```

