This changes your working directory to the shiny-new-feature branch. Keep any changes in this branch specific to one bug or feature so it is clear what the branch brings to *pandas*. You can have many shiny-new-features and switch in between them using the git checkout command.

When creating this branch, make sure your master branch is up to date with the latest upstream master version. To update your local master branch, you can do:

```
git checkout master
git pull upstream master --ff-only
```

When you want to update the feature branch with changes in master after you created the branch, check the section on *updating a PR*.

4.1.4 Contributing to the documentation

Contributing to the documentation benefits everyone who uses *pandas*. We encourage you to help us improve the documentation, and you don't have to be an expert on *pandas* to do so! In fact, there are sections of the docs that are worse off after being written by experts. If something in the docs doesn't make sense to you, updating the relevant section after you figure it out is a great way to ensure it will help the next person.

Documentation:

- *About the* pandas *documentation*
- *Updating a* pandas *docstring*
- *How to build the* pandas *documentation*
  - *Requirements*
  - *Building the documentation*
  - *Building master branch documentation*

About the *pandas* documentation

The documentation is written in **reStructuredText**, which is almost like writing in plain English, and built using Sphinx. The Sphinx Documentation has an excellent introduction to reST. Review the Sphinx docs to perform more complex changes to the documentation as well.

Some other important things to know about the docs:

- The *pandas* documentation consists of two parts: the docstrings in the code itself and the docs in this folder `doc/`.

  The docstrings provide a clear explanation of the usage of the individual functions, while the documentation in this folder consists of tutorial-like overviews per topic together with some other information (what's new, installation, etc).

- The docstrings follow a pandas convention, based on the **Numpy Docstring Standard**. Follow the *pandas docstring guide* for detailed instructions on how to write a correct docstring.

pandas docstring guide

About docstrings and standards

A Python docstring is a string used to document a Python module, class, function or method, so programmers can understand what it does without having to read the details of the implementation.

Also, it is a common practice to generate online (html) documentation automatically from docstrings. Sphinx serves this purpose.

Next example gives an idea on how a docstring looks like:

```
def add(num1, num2):
    """
    Add up two integer numbers.

    This function simply wraps the `+` operator, and does not
    do anything interesting, except for illustrating what is
    the docstring of a very simple function.

    Parameters
    ----------
    num1 : int
        First number to add
    num2 : int
        Second number to add

    Returns
    -------
    int
        The sum of `num1` and `num2`

    See Also
    --------
    subtract : Subtract one integer from another

    Examples
    --------
    >>> add(2, 2)
    4
    >>> add(25, 0)
    25
    >>> add(10, -10)
    0
    """
    return num1 + num2
```

Some standards exist about docstrings, so they are easier to read, and they can be exported to other formats such as html or pdf.

The first conventions every Python docstring should follow are defined in PEP-257.

As PEP-257 is quite open, and some other standards exist on top of it. In the case of pandas, the numpy docstring convention is followed. The conventions is explained in this document:

- numpydoc docstring guide (which is based in the original Guide to NumPy/SciPy documentation)

numpydoc is a Sphinx extension to support the numpy docstring convention.

The standard uses reStructuredText (reST). reStructuredText is a markup language that allows encoding styles in plain text files. Documentation about reStructuredText can be found in:

- Sphinx reStructuredText primer
- Quick reStructuredText reference
- Full reStructuredText specification

Pandas has some helpers for sharing docstrings between related classes, see *Sharing docstrings*.

The rest of this document will summarize all the above guides, and will provide additional convention specific to the pandas project.

Writing a docstring

General rules

Docstrings must be defined with three double-quotes. No blank lines should be left before or after the docstring. The text starts in the next line after the opening quotes. The closing quotes have their own line (meaning that they are not at the end of the last sentence).

In rare occasions reST styles like bold text or italics will be used in docstrings, but is it common to have inline code, which is presented between backticks. It is considered inline code:

- The name of a parameter
- Python code, a module, function, built-in, type, literal... (e.g. `os`, `list`, `numpy.abs`, `datetime.date`, `True`)
- A pandas class (in the form `:class:`pandas.Series``)
- A pandas method (in the form `:meth:`pandas.Series.sum``)
- A pandas function (in the form `:func:`pandas.to_datetime``)

Note: To display only the last component of the linked class, method or function, prefix it with ~. For example, `:class:`~pandas.Series`` will link to `pandas.Series` but only display the last part, `Series` as the link text. See Sphinx cross-referencing syntax for details.

Good:

```
def add_values(arr):
    """
    Add the values in `arr`.

    This is equivalent to Python `sum` of :meth:`pandas.Series.sum`.

    Some sections are omitted here for simplicity.
    """
    return sum(arr)
```

Bad:

```
def func():

    """Some function.

```

(continues on next page)

(continued from previous page)

```
    With several mistakes in the docstring.

    It has a blank like after the signature `def func():`.

    The text 'Some function' should go in the line after the
    opening quotes of the docstring, not in the same line.

    There is a blank line between the docstring and the first line
    of code `foo = 1`.

    The closing quotes should be in the next line, not in this one."""

    foo = 1
    bar = 2
    return foo + bar
```

Section 1: Short summary

The short summary is a single sentence that expresses what the function does in a concise way.

The short summary must start with a capital letter, end with a dot, and fit in a single line. It needs to express what the object does without providing details. For functions and methods, the short summary must start with an infinitive verb.

Good:

```
def astype(dtype):
    """
    Cast Series type.

    This section will provide further details.
    """
    pass
```

Bad:

```
def astype(dtype):
    """
    Casts Series type.

    Verb in third-person of the present simple, should be infinitive.
    """
    pass
```

```
def astype(dtype):
    """
    Method to cast Series type.

    Does not start with verb.
    """
    pass
```

```
def astype(dtype):
    """
```

(continues on next page)

(continued from previous page)

```
    Cast Series type

    Missing dot at the end.
    """
    pass
```

```
def astype(dtype):
    """
    Cast Series type from its current type to the new type defined in
    the parameter dtype.

    Summary is too verbose and doesn't fit in a single line.
    """
    pass
```

Section 2: Extended summary

The extended summary provides details on what the function does. It should not go into the details of the parameters, or discuss implementation notes, which go in other sections.

A blank line is left between the short summary and the extended summary. And every paragraph in the extended summary is finished by a dot.

The extended summary should provide details on why the function is useful and their use cases, if it is not too generic.

```
def unstack():
    """
    Pivot a row index to columns.

    When using a MultiIndex, a level can be pivoted so each value in
    the index becomes a column. This is especially useful when a subindex
    is repeated for the main index, and data is easier to visualize as a
    pivot table.

    The index level will be automatically removed from the index when added
    as columns.
    """
    pass
```

Section 3: Parameters

The details of the parameters will be added in this section. This section has the title "Parameters", followed by a line with a hyphen under each letter of the word "Parameters". A blank line is left before the section title, but not after, and not between the line with the word "Parameters" and the one with the hyphens.

After the title, each parameter in the signature must be documented, including **args* and ***kwargs*, but not *self*.

The parameters are defined by their name, followed by a space, a colon, another space, and the type (or types). Note that the space between the name and the colon is important. Types are not defined for **args* and ***kwargs*, but must be defined for all other parameters. After the parameter definition, it is required to have a line with the parameter description, which is indented, and can have multiple lines. The description must start with a capital letter, and finish with a dot.

For keyword arguments with a default value, the default will be listed after a comma at the end of the type. The exact form of the type in this case will be "int, default 0". In some cases it may be useful to explain what the default argument means, which can be added after a comma "int, default -1, meaning all cpus".

In cases where the default value is *None*, meaning that the value will not be used. Instead of "str, default None", it is preferred to write "str, optional". When *None* is a value being used, we will keep the form "str, default None". For example, in *df.to_csv(compression=None)*, *None* is not a value being used, but means that compression is optional, and no compression is being used if not provided. In this case we will use *str, optional*. Only in cases like *func(value=None)* and *None* is being used in the same way as *0* or *foo* would be used, then we will specify "str, int or None, default None".

Good:

```
class Series:
    def plot(self, kind, color='blue', **kwargs):
        """
        Generate a plot.

        Render the data in the Series as a matplotlib plot of the
        specified kind.

        Parameters
        ----------
        kind : str
            Kind of matplotlib plot.
        color : str, default 'blue'
            Color name or rgb code.
        **kwargs
            These parameters will be passed to the matplotlib plotting
            function.
        """
        pass
```

Bad:

```
class Series:
    def plot(self, kind, **kwargs):
        """
        Generate a plot.

        Render the data in the Series as a matplotlib plot of the
        specified kind.

        Note the blank line between the parameters title and the first
        parameter. Also, note that after the name of the parameter `kind`
        and before the colon, a space is missing.

        Also, note that the parameter descriptions do not start with a
        capital letter, and do not finish with a dot.

        Finally, the `**kwargs` parameter is missing.

        Parameters
        ----------

        kind: str
            kind of matplotlib plot
        """
```

(continues on next page)

(continued from previous page)

```
        pass
```

Parameter types

When specifying the parameter types, Python built-in data types can be used directly (the Python type is preferred to the more verbose string, integer, boolean, etc):

- int
- float
- str
- bool

For complex types, define the subtypes. For *dict* and *tuple*, as more than one type is present, we use the brackets to help read the type (curly brackets for *dict* and normal brackets for *tuple*):

- list of int
- dict of {str : int}
- tuple of (str, int, int)
- tuple of (str,)
- set of str

In case where there are just a set of values allowed, list them in curly brackets and separated by commas (followed by a space). If the values are ordinal and they have an order, list them in this order. Otherwise, list the default value first, if there is one:

- {0, 10, 25}
- {'simple', 'advanced'}
- {'low', 'medium', 'high'}
- {'cat', 'dog', 'bird'}

If the type is defined in a Python module, the module must be specified:

- datetime.date
- datetime.datetime
- decimal.Decimal

If the type is in a package, the module must be also specified:

- numpy.ndarray
- scipy.sparse.coo_matrix

If the type is a pandas type, also specify pandas except for Series and DataFrame:

- Series
- DataFrame
- pandas.Index
- pandas.Categorical
- pandas.arrays.SparseArray

If the exact type is not relevant, but must be compatible with a numpy array, array-like can be specified. If Any type that can be iterated is accepted, iterable can be used:

- array-like
- iterable

If more than one type is accepted, separate them by commas, except the last two types, that need to be separated by the word 'or':

- int or float
- float, decimal.Decimal or None
- str or list of str

If `None` is one of the accepted values, it always needs to be the last in the list.

For axis, the convention is to use something like:

- axis : {0 or 'index', 1 or 'columns', None}, default None

Section 4: Returns or Yields

If the method returns a value, it will be documented in this section. Also if the method yields its output.

The title of the section will be defined in the same way as the "Parameters". With the names "Returns" or "Yields" followed by a line with as many hyphens as the letters in the preceding word.

The documentation of the return is also similar to the parameters. But in this case, no name will be provided, unless the method returns or yields more than one value (a tuple of values).

The types for "Returns" and "Yields" are the same as the ones for the "Parameters". Also, the description must finish with a dot.

For example, with a single value:

```
def sample():
    """
    Generate and return a random number.

    The value is sampled from a continuous uniform distribution between
    0 and 1.

    Returns
    -------
    float
        Random number generated.
    """
    return np.random.random()
```

With more than one value:

```
import string

def random_letters():
    """
    Generate and return a sequence of random letters.

    The length of the returned string is also random, and is also
    returned.
```

(continues on next page)

(continued from previous page)

```
    Returns
    -------
    length : int
        Length of the returned string.
    letters : str
        String of random letters.
    """
    length = np.random.randint(1, 10)
    letters = ''.join(np.random.choice(string.ascii_lowercase)
                      for i in range(length))
    return length, letters
```

If the method yields its value:

```
def sample_values():
    """
    Generate an infinite sequence of random numbers.

    The values are sampled from a continuous uniform distribution between
    0 and 1.

    Yields
    ------
    float
        Random number generated.
    """
    while True:
        yield np.random.random()
```

Section 5: See Also

This section is used to let users know about pandas functionality related to the one being documented. In rare cases, if no related methods or functions can be found at all, this section can be skipped.

An obvious example would be the *head()* and *tail()* methods. As *tail()* does the equivalent as *head()* but at the end of the *Series* or *DataFrame* instead of at the beginning, it is good to let the users know about it.

To give an intuition on what can be considered related, here there are some examples:

- `loc` and `iloc`, as they do the same, but in one case providing indices and in the other positions
- `max` and `min`, as they do the opposite
- `iterrows`, `itertuples` and `items`, as it is easy that a user looking for the method to iterate over columns ends up in the method to iterate over rows, and vice-versa
- `fillna` and `dropna`, as both methods are used to handle missing values
- `read_csv` and `to_csv`, as they are complementary
- `merge` and `join`, as one is a generalization of the other
- `astype` and `pandas.to_datetime`, as users may be reading the documentation of `astype` to know how to cast as a date, and the way to do it is with `pandas.to_datetime`
- `where` is related to `numpy.where`, as its functionality is based on it

When deciding what is related, you should mainly use your common sense and think about what can be useful for the users reading the documentation, especially the less experienced ones.

When relating to other libraries (mainly `numpy`), use the name of the module first (not an alias like `np`). If the function is in a module which is not the main one, like `scipy.sparse`, list the full module (e.g. `scipy.sparse.coo_matrix`).

This section, as the previous, also has a header, "See Also" (note the capital S and A). Also followed by the line with hyphens, and preceded by a blank line.

After the header, we will add a line for each related method or function, followed by a space, a colon, another space, and a short description that illustrated what this method or function does, why is it relevant in this context, and what are the key differences between the documented function and the one referencing. The description must also finish with a dot.

Note that in "Returns" and "Yields", the description is located in the following line than the type. But in this section it is located in the same line, with a colon in between. If the description does not fit in the same line, it can continue in the next ones, but it has to be indented in them.

For example:

```
class Series:
    def head(self):
        """
        Return the first 5 elements of the Series.

        This function is mainly useful to preview the values of the
        Series without displaying the whole of it.

        Returns
        -------
        Series
            Subset of the original series with the 5 first values.

        See Also
        --------
        Series.tail : Return the last 5 elements of the Series.
        Series.iloc : Return a slice of the elements in the Series,
            which can also be used to return the first or last n.
        """
        return self.iloc[:5]
```

Section 6: Notes

This is an optional section used for notes about the implementation of the algorithm. Or to document technical aspects of the function behavior.

Feel free to skip it, unless you are familiar with the implementation of the algorithm, or you discover some counter-intuitive behavior while writing the examples for the function.

This section follows the same format as the extended summary section.

Section 7: Examples

This is one of the most important sections of a docstring, even if it is placed in the last position. As often, people understand concepts better with examples, than with accurate explanations.

Examples in docstrings, besides illustrating the usage of the function or method, must be valid Python code, that in a deterministic way returns the presented output, and that can be copied and run by users.

They are presented as a session in the Python terminal. `>>>` is used to present code. `...` is used for code continuing from the previous line. Output is presented immediately after the last line of code generating the output (no blank lines in between). Comments describing the examples can be added with blank lines before and after them.

The way to present examples is as follows:

1. Import required libraries (except `numpy` and `pandas`)
2. Create the data required for the example
3. Show a very basic example that gives an idea of the most common use case
4. Add examples with explanations that illustrate how the parameters can be used for extended functionality

A simple example could be:

```
class Series:

    def head(self, n=5):
        """
        Return the first elements of the Series.

        This function is mainly useful to preview the values of the
        Series without displaying the whole of it.

        Parameters
        ----------
        n : int
            Number of values to return.

        Return
        ------
        pandas.Series
            Subset of the original series with the n first values.

        See Also
        --------
        tail : Return the last n elements of the Series.

        Examples
        --------
        >>> s = pd.Series(['Ant', 'Bear', 'Cow', 'Dog', 'Falcon',
        ...                'Lion', 'Monkey', 'Rabbit', 'Zebra'])
        >>> s.head()
        0   Ant
        1   Bear
        2   Cow
        3   Dog
        4   Falcon
        dtype: object
```

(continues on next page)

(continued from previous page)

```
        With the `n` parameter, we can change the number of returned rows:

        >>> s.head(n=3)
        0   Ant
        1   Bear
        2   Cow
        dtype: object
        """
        return self.iloc[:n]
```

The examples should be as concise as possible. In cases where the complexity of the function requires long examples, is recommended to use blocks with headers in bold. Use double star `**` to make a text bold, like in `**this example**`.

Conventions for the examples

Code in examples is assumed to always start with these two lines which are not shown:

```
import numpy as np
import pandas as pd
```

Any other module used in the examples must be explicitly imported, one per line (as recommended in **PEP 8#imports**) and avoiding aliases. Avoid excessive imports, but if needed, imports from the standard library go first, followed by third-party libraries (like matplotlib).

When illustrating examples with a single `Series` use the name `s`, and if illustrating with a single `DataFrame` use the name `df`. For indices, `idx` is the preferred name. If a set of homogeneous `Series` or `DataFrame` is used, name them `s1`, `s2`, `s3...` or `df1`, `df2`, `df3...` If the data is not homogeneous, and more than one structure is needed, name them with something meaningful, for example `df_main` and `df_to_join`.

Data used in the example should be as compact as possible. The number of rows is recommended to be around 4, but make it a number that makes sense for the specific example. For example in the `head` method, it requires to be higher than 5, to show the example with the default values. If doing the `mean`, we could use something like `[1, 2, 3]`, so it is easy to see that the value returned is the mean.

For more complex examples (grouping for example), avoid using data without interpretation, like a matrix of random numbers with columns A, B, C, D... And instead use a meaningful example, which makes it easier to understand the concept. Unless required by the example, use names of animals, to keep examples consistent. And numerical properties of them.

When calling the method, keywords arguments `head(n=3)` are preferred to positional arguments `head(3)`.

Good:

```
class Series:

    def mean(self):
        """
        Compute the mean of the input.

        Examples
        --------
        >>> s = pd.Series([1, 2, 3])
        >>> s.mean()
        2
        """
```

(continues on next page)

(continued from previous page)

```
        pass


    def fillna(self, value):
        """
        Replace missing values by `value`.

        Examples
        --------
        >>> s = pd.Series([1, np.nan, 3])
        >>> s.fillna(0)
        [1, 0, 3]
        """
        pass

    def groupby_mean(self):
        """
        Group by index and return mean.

        Examples
        --------
        >>> s = pd.Series([380., 370., 24., 26],
        ...               name='max_speed',
        ...               index=['falcon', 'falcon', 'parrot', 'parrot'])
        >>> s.groupby_mean()
        index
        falcon    375.0
        parrot     25.0
        Name: max_speed, dtype: float64
        """
        pass

    def contains(self, pattern, case_sensitive=True, na=numpy.nan):
        """
        Return whether each value contains `pattern`.

        In this case, we are illustrating how to use sections, even
        if the example is simple enough and does not require them.

        Examples
        --------
        >>> s = pd.Series('Antelope', 'Lion', 'Zebra', np.nan)
        >>> s.contains(pattern='a')
        0    False
        1    False
        2     True
        3      NaN
        dtype: bool

        **Case sensitivity**

        With `case_sensitive` set to `False` we can match `a` with both
        `a` and `A`:

        >>> s.contains(pattern='a', case_sensitive=False)
        0     True
        1    False
```

(continues on next page)

(continued from previous page)

```
        2     True
        3      NaN
        dtype: bool

        **Missing values**

        We can fill missing values in the output using the `na` parameter:

        >>> s.contains(pattern='a', na=False)
        0    False
        1    False
        2     True
        3    False
        dtype: bool
        """
        pass
```

Bad:

```
def method(foo=None, bar=None):
    """
    A sample DataFrame method.

    Do not import numpy and pandas.

    Try to use meaningful data, when it makes the example easier
    to understand.

    Try to avoid positional arguments like in `df.method(1)`. They
    can be all right if previously defined with a meaningful name,
    like in `present_value(interest_rate)`, but avoid them otherwise.

    When presenting the behavior with different parameters, do not place
    all the calls one next to the other. Instead, add a short sentence
    explaining what the example shows.

    Examples
    --------
    >>> import numpy as np
    >>> import pandas as pd
    >>> df = pd.DataFrame(np.random.randn(3, 3),
    ...                   columns=('a', 'b', 'c'))
    >>> df.method(1)
    21
    >>> df.method(bar=14)
    123
    """
    pass
```

Tips for getting your examples pass the doctests

Getting the examples pass the doctests in the validation script can sometimes be tricky. Here are some attention points:

- Import all needed libraries (except for pandas and numpy, those are already imported as `import pandas as pd` and `import numpy as np`) and define all variables you use in the example.
- Try to avoid using random data. However random data might be OK in some cases, like if the function you are documenting deals with probability distributions, or if the amount of data needed to make the function result meaningful is too much, such that creating it manually is very cumbersome. In those cases, always use a fixed random seed to make the generated examples predictable. Example:

```
>>> np.random.seed(42)
>>> df = pd.DataFrame({'normal': np.random.normal(100, 5, 20)})
```

- If you have a code snippet that wraps multiple lines, you need to use '...' on the continued lines:

```
>>> df = pd.DataFrame([[1, 2, 3], [4, 5, 6]], index=['a', 'b', 'c'],
...                   columns=['A', 'B'])
```

- If you want to show a case where an exception is raised, you can do:

```
>>> pd.to_datetime(["712-01-01"])
Traceback (most recent call last):
OutOfBoundsDatetime: Out of bounds nanosecond timestamp: 712-01-01 00:00:00
```

  It is essential to include the "Traceback (most recent call last):", but for the actual error only the error name is sufficient.

- If there is a small part of the result that can vary (e.g. a hash in an object representation), you can use `...` to represent this part.

  If you want to show that `s.plot()` returns a matplotlib AxesSubplot object, this will fail the doctest

```
>>> s.plot()
<matplotlib.axes._subplots.AxesSubplot at 0x7efd0c0b0690>
```

  However, you can do (notice the comment that needs to be added)

```
>>> s.plot()
<matplotlib.axes._subplots.AxesSubplot at ...>
```

Plots in examples

There are some methods in pandas returning plots. To render the plots generated by the examples in the documentation, the `.. plot::` directive exists.

To use it, place the next code after the "Examples" header as shown below. The plot will be generated automatically when building the documentation.

```
class Series:
    def plot(self):
        """
        Generate a plot with the `Series` data.
```

(continues on next page)

(continued from previous page)

```
        Examples
        --------

        .. plot::
            :context: close-figs

            >>> s = pd.Series([1, 2, 3])
            >>> s.plot()
        """
        pass
```

Sharing docstrings

Pandas has a system for sharing docstrings, with slight variations, between classes. This helps us keep docstrings consistent, while keeping things clear for the user reading. It comes at the cost of some complexity when writing.

Each shared docstring will have a base template with variables, like `%(klass)s`. The variables filled in later on using the `Substitution` decorator. Finally, docstrings can be appended to with the `Appender` decorator.

In this example, we'll create a parent docstring normally (this is like `pandas.core.generic.NDFrame`. Then we'll have two children (like `pandas.core.series.Series` and `pandas.core.frame.DataFrame`). We'll substitute the children's class names in this docstring.

```
class Parent:
    def my_function(self):
        """Apply my function to %(klass)s."""
        ...


class ChildA(Parent):
    @Substitution(klass="ChildA")
    @Appender(Parent.my_function.__doc__)
    def my_function(self):
        ...


class ChildB(Parent):
    @Substitution(klass="ChildB")
    @Appender(Parent.my_function.__doc__)
    def my_function(self):
        ...
```

The resulting docstrings are

```
>>> print(Parent.my_function.__doc__)
Apply my function to %(klass)s.
>>> print(ChildA.my_function.__doc__)
Apply my function to ChildA.
>>> print(ChildB.my_function.__doc__)
Apply my function to ChildB.
```

Notice two things:

1. We "append" the parent docstring to the children docstrings, which are initially empty.
2. Python decorators are applied inside out. So the order is Append then Substitution, even though Substitution comes first in the file.

Our files will often contain a module-level `_shared_doc_kwargs` with some common substitution values (things like `klass`, `axes`, etc).

You can substitute and append in one shot with something like

```
@Appender(template % _shared_doc_kwargs)
def my_function(self):
    ...
```

where `template` may come from a module-level `_shared_docs` dictionary mapping function names to docstrings. Wherever possible, we prefer using `Appender` and `Substitution`, since the docstring-writing processes is slightly closer to normal.

See `pandas.core.generic.NDFrame.fillna` for an example template, and `pandas.core.series.Series.fillna` and `pandas.core.generic.frame.fillna` for the filled versions.

- The tutorials make heavy use of the ipython directive sphinx extension. This directive lets you put code in the documentation which will be run during the doc build. For example:

```
.. ipython:: python

    x = 2
    x**3
```

will be rendered as:

```
In [1]: x = 2

In [2]: x**3
Out[2]: 8
```

Almost all code examples in the docs are run (and the output saved) during the doc build. This approach means that code examples will always be up to date, but it does make the doc building a bit more complex.

- Our API documentation in `doc/source/api.rst` houses the auto-generated documentation from the docstrings. For classes, there are a few subtleties around controlling which methods and attributes have pages auto-generated.

  We have two autosummary templates for classes.

  1. `_templates/autosummary/class.rst`. Use this when you want to automatically generate a page for every public method and attribute on the class. The `Attributes` and `Methods` sections will be automatically added to the class' rendered documentation by numpydoc. See `DataFrame` for an example.
  2. `_templates/autosummary/class_without_autosummary`. Use this when you want to pick a subset of methods / attributes to auto-generate pages for. When using this template, you should include an `Attributes` and `Methods` section in the class docstring. See `CategoricalIndex` for an example.

  Every method should be included in a `toctree` in `api.rst`, else Sphinx will emit a warning.

Note: The `.rst` files are used to automatically generate Markdown and HTML versions of the docs. For this reason, please do not edit `CONTRIBUTING.md` directly, but instead make any changes to `doc/source/development/contributing.rst`. Then, to generate `CONTRIBUTING.md`, use pandoc with the following command:

```
pandoc doc/source/development/contributing.rst -t markdown_github > CONTRIBUTING.md
```

The utility script `scripts/validate_docstrings.py` can be used to get a csv summary of the API documentation. And also validate common errors in the docstring of a specific class, function or method. The summary also compares the list of methods documented in `doc/source/api.rst` (which is used to generate the API Reference page) and the actual public methods. This will identify methods documented in `doc/source/api.rst` that are not actually class methods, and existing methods that are not documented in `doc/source/api.rst`.

Updating a *pandas* docstring

When improving a single function or method's docstring, it is not necessarily needed to build the full documentation (see next section). However, there is a script that checks a docstring (for example for the `DataFrame.mean` method):

```
python scripts/validate_docstrings.py pandas.DataFrame.mean
```

This script will indicate some formatting errors if present, and will also run and test the examples included in the docstring. Check the *pandas docstring guide* for a detailed guide on how to format the docstring.

The examples in the docstring ('doctests') must be valid Python code, that in a deterministic way returns the presented output, and that can be copied and run by users. This can be checked with the script above, and is also tested on Travis. A failing doctest will be a blocker for merging a PR. Check the *examples* section in the docstring guide for some tips and tricks to get the doctests passing.

When doing a PR with a docstring update, it is good to post the output of the validation script in a comment on github.

How to build the *pandas* documentation

Requirements

First, you need to have a development environment to be able to build pandas (see the docs on *creating a development environment above*).

Building the documentation

So how do you build the docs? Navigate to your local `doc/` directory in the console and run:

```
python make.py html
```

Then you can find the HTML output in the folder `doc/build/html/`.

The first time you build the docs, it will take quite a while because it has to run all the code examples and build all the generated docstring pages. In subsequent evocations, sphinx will try to only build the pages that have been modified.

If you want to do a full clean build, do:

```
python make.py clean
python make.py html
```

You can tell `make.py` to compile only a single section of the docs, greatly reducing the turn-around time for checking your changes.

```
# omit autosummary and API section
python make.py clean
python make.py --no-api

# compile the docs with only a single section, relative to the "source" folder.
```

(continues on next page)

(continued from previous page)

```
# For example, compiling only this guide (doc/source/development/contributing.rst)
python make.py clean
python make.py --single development/contributing.rst

# compile the reference docs for a single function
python make.py clean
python make.py --single pandas.DataFrame.join
```

For comparison, a full documentation build may take 15 minutes, but a single section may take 15 seconds. Subsequent builds, which only process portions you have changed, will be faster.

You can also specify to use multiple cores to speed up the documentation build:

```
python make.py html --num-jobs 4
```

Open the following file in a web browser to see the full documentation you just built:

```
doc/build/html/index.html
```

And you'll have the satisfaction of seeing your new and improved documentation!

Building master branch documentation

When pull requests are merged into the *pandas* `master` branch, the main parts of the documentation are also built by Travis-CI. These docs are then hosted here, see also the *Continuous Integration* section.

4.1.5 Contributing to the code base

Code Base:

Code standards

Writing good code is not just about what you write. It is also about *how* you write it. During *Continuous Integration* testing, several tools will be run to check your code for stylistic errors. Generating any warnings will cause the test to fail. Thus, good style is a requirement for submitting code to *pandas*.

There is a tool in pandas to help contributors verify their changes before contributing them to the project:

```
./ci/code_checks.sh
```

The script verifies the linting of code files, it looks for common mistake patterns (like missing spaces around sphinx directives that make the documentation not being rendered properly) and it also validates the doctests. It is possible to run the checks independently by using the parameters `lint`, `patterns` and `doctests` (e.g. `./ci/code_checks.sh lint`).

In addition, because a lot of people use our library, it is important that we do not make sudden changes to the code that could have the potential to break a lot of user code as a result, that is, we need it to be as *backwards compatible* as possible to avoid mass breakages.

Additional standards are outlined on the pandas code style guide

Optional dependencies

Optional dependencies (e.g. matplotlib) should be imported with the private helper `pandas.compat._optional.import_optional_dependency`. This ensures a consistent error message when the dependency is not met.

All methods using an optional dependency should include a test asserting that an `ImportError` is raised when the optional dependency is not found. This test should be skipped if the library is present.

All optional dependencies should be documented in *Optional dependencies* and the minimum required version should be set in the `pandas.compat._optional.VERSIONS` dict.

C (cpplint)

pandas uses the Google standard. Google provides an open source style checker called `cpplint`, but we use a fork of it that can be found here. Here are *some* of the more common `cpplint` issues:

- we restrict line-length to 80 characters to promote readability
- every header file must include a header guard to avoid name collisions if re-included

Continuous Integration will run the cpplint tool and report any stylistic errors in your code. Therefore, it is helpful before submitting code to run the check yourself:

```
cpplint --extensions=c,h --headers=h --filter=-readability/casting,-runtime/int,-
↪build/include_subdir modified-c-file
```

You can also run this command on an entire directory if necessary:

```
cpplint --extensions=c,h --headers=h --filter=-readability/casting,-runtime/int,-
↪build/include_subdir --recursive modified-c-directory
```

To make your commits compliant with this standard, you can install the ClangFormat tool, which can be downloaded here. To configure, in your home directory, run the following command:

```
clang-format style=google -dump-config  > .clang-format
```

Then modify the file to ensure that any indentation width parameters are at least four. Once configured, you can run the tool as follows:

```
clang-format modified-c-file
```

This will output what your file will look like if the changes are made, and to apply them, run the following command:

```
clang-format -i modified-c-file
```

To run the tool on an entire directory, you can run the following analogous commands:

```
clang-format modified-c-directory/*.c modified-c-directory/*.h
clang-format -i modified-c-directory/*.c modified-c-directory/*.h
```

Do note that this tool is best-effort, meaning that it will try to correct as many errors as possible, but it may not correct *all* of them. Thus, it is recommended that you run `cpplint` to double check and make any other style fixes manually.

Python (PEP8 / black)

pandas follows the PEP8 standard and uses Black and Flake8 to ensure a consistent code format throughout the project.

Continuous Integration will run those tools and report any stylistic errors in your code. Therefore, it is helpful before submitting code to run the check yourself:

```
black pandas
git diff upstream/master -u -- "*.py" | flake8 --diff
```

to auto-format your code. Additionally, many editors have plugins that will apply `black` as you edit files.

You should use a `black` version >= 19.10b0 as previous versions are not compatible with the pandas codebase.

Optionally, you may wish to setup pre-commit hooks to automatically run `black` and `flake8` when you make a git commit. This can be done by installing `pre-commit`:

```
pip install pre-commit
```

and then running:

```
pre-commit install
```

from the root of the pandas repository. Now `black` and `flake8` will be run each time you commit changes. You can skip these checks with `git commit --no-verify`.

One caveat about `git diff upstream/master -u -- "*.py" | flake8 --diff`: this command will catch any stylistic errors in your changes specifically, but be beware it may not catch all of them. For example, if you delete the only usage of an imported function, it is stylistically incorrect to import an unused function. However, style-checking the diff will not catch this because the actual import is not part of the diff. Thus, for completeness, you should run this command, though it will take longer:

```
git diff upstream/master --name-only -- "*.py" | xargs -r flake8
```

Note that on OSX, the `-r` flag is not available, so you have to omit it and run this slightly modified command:

```
git diff upstream/master --name-only -- "*.py" | xargs flake8
```

Windows does not support the `xargs` command (unless installed for example via the MinGW toolchain), but one can imitate the behaviour as follows:

```
for /f %i in ('git diff upstream/master --name-only -- "*.py"') do flake8 %i
```

This will get all the files being changed by the PR (and ending with `.py`), and run `flake8` on them, one after the other.

Import formatting

pandas uses isort to standardise import formatting across the codebase.

A guide to import layout as per pep8 can be found here.

A summary of our current import sections (in order):

- Future
- Python Standard Library
- Third Party
- `pandas._libs`, `pandas.compat`, `pandas.util._*`, `pandas.errors` (largely not dependent on `pandas.core`)
- `pandas.core.dtypes` (largely not dependent on the rest of `pandas.core`)
- Rest of `pandas.core.*`
- Non-core `pandas.io`, `pandas.plotting`, `pandas.tseries`
- Local application/library specific imports

Imports are alphabetically sorted within these sections.

As part of *Continuous Integration* checks we run:

```
isort --recursive --check-only pandas
```

to check that imports are correctly formatted as per the *setup.cfg*.

If you see output like the below in *Continuous Integration* checks:

```
Check import format using isort
ERROR: /home/travis/build/pandas-dev/pandas/pandas/io/pytables.py Imports are␣
↪incorrectly sorted
Check import format using isort DONE
The command "ci/code_checks.sh" exited with 1
```

You should run:

```
isort pandas/io/pytables.py
```

to automatically format imports correctly. This will modify your local copy of the files.

The –*recursive* flag can be passed to sort all files in a directory.

You can then verify the changes look ok, then git *commit* and *push*.

Backwards compatibility

Please try to maintain backward compatibility. *pandas* has lots of users with lots of existing code, so don't break it if at all possible. If you think breakage is required, clearly state why as part of the pull request. Also, be careful when changing method signatures and add deprecation warnings where needed. Also, add the deprecated sphinx directive to the deprecated functions or methods.

If a function with the same arguments as the one being deprecated exist, you can use the `pandas.util._decorators.deprecate`:

```
from pandas.util._decorators import deprecate

deprecate('old_func', 'new_func', '0.21.0')
```

Otherwise, you need to do it manually:

```
import warnings


def old_func():
    """Summary of the function.

    .. deprecated:: 0.21.0
       Use new_func instead.
    """
    warnings.warn('Use new_func instead.', FutureWarning, stacklevel=2)
    new_func()


def new_func():
    pass
```

You'll also need to

1. Write a new test that asserts a warning is issued when calling with the deprecated argument
2. Update all of pandas existing tests and code to use the new argument

See *Testing warnings* for more.

Type Hints

pandas strongly encourages the use of **PEP 484** style type hints. New development should contain type hints and pull requests to annotate existing code are accepted as well!

Style Guidelines

Types imports should follow the `from typing import ...` convention. So rather than

```
import typing

primes: typing.List[int] = []
```

You should write

```
from typing import List, Optional, Union

primes: List[int] = []
```

`Optional` should be used where applicable, so instead of

```
maybe_primes: List[Union[int, None]] = []
```

You should write

```
maybe_primes: List[Optional[int]] = []
```

In some cases in the code base classes may define class variables that shadow builtins. This causes an issue as described in Mypy 1775. The defensive solution here is to create an unambiguous alias of the builtin and use that without your annotation. For example, if you come across a definition like

```
class SomeClass1:
    str = None
```

The appropriate way to annotate this would be as follows

```
str_type = str

class SomeClass2:
    str: str_type = None
```

In some cases you may be tempted to use `cast` from the typing module when you know better than the analyzer. This occurs particularly when using custom inference functions. For example

```
from typing import cast

from pandas.core.dtypes.common import is_number

def cannot_infer_bad(obj: Union[str, int, float]):

    if is_number(obj):
        ...
    else:  # Reasonably only str objects would reach this but...
        obj = cast(str, obj)  # Mypy complains without this!
        return obj.upper()
```

The limitation here is that while a human can reasonably understand that `is_number` would catch the `int` and `float` types mypy cannot make that same inference just yet (see mypy #5206. While the above works, the use of `cast` is **strongly discouraged**. Where applicable a refactor of the code to appease static analysis is preferable

```
def cannot_infer_good(obj: Union[str, int, float]):

    if isinstance(obj, str):
        return obj.upper()
    else:
        ...
```

With custom types and inference this is not always possible so exceptions are made, but every effort should be exhausted to avoid `cast` before going down such paths.

Pandas-specific Types

Commonly used types specific to *pandas* will appear in pandas._typing and you should use these where applicable. This module is private for now but ultimately this should be exposed to third party libraries who want to implement type checking against pandas.

For example, quite a few functions in *pandas* accept a `dtype` argument. This can be expressed as a string like `"object"`, a `numpy.dtype` like `np.int64` or even a pandas `ExtensionDtype` like `pd.CategoricalDtype`. Rather than burden the user with having to constantly annotate all of those options, this can simply be imported and reused from the pandas._typing module

```
from pandas._typing import Dtype

def as_type(dtype: Dtype) -> ...:
    ...
```

This module will ultimately house types for repeatedly used concepts like “path-like”, “array-like”, “numeric”, etc… and can also hold aliases for commonly appearing parameters like *axis*. Development of this module is active so be sure to refer to the source for the most up to date list of available types.

Validating Type Hints

pandas uses mypy to statically analyze the code base and type hints. After making any change you can ensure your type hints are correct by running

```
mypy pandas
```

Testing with continuous integration

The *pandas* test suite will run automatically on Travis-CI and Azure Pipelines continuous integration services, once your pull request is submitted. However, if you wish to run the test suite on a branch prior to submitting the pull request, then the continuous integration services need to be hooked to your GitHub repository. Instructions are here for Travis-CI and Azure Pipelines.

A pull-request will be considered for merging when you have an all ‘green’ build. If any tests are failing, then you will get a red ‘X’, where you can click through to see the individual failed tests. This is an example of a green build.