Working with missing data

In this section, we will discuss missing (also referred to as NA) values in pandas.

> **Note:** The choice of using `NaN` internally to denote missing data was largely for simplicity and performance reasons. It differs from the MaskedArray approach of, for example, **`scikits.timeseries`**. We are hopeful that NumPy will soon be able to provide a native NA type solution (similar to R) performant enough to be used in pandas.

See the *cookbook* for some advanced strategies

Missing data basics

When / why does data become missing?

Some might quibble over our usage of *missing*. By "missing" we simply mean **null** or "not present for whatever reason". Many data sets simply arrive with missing data, either because it exists and was not collected or it never existed. For example, in a collection of financial time series, some of the time series might start on different dates. Thus, values prior to the start date would generally be marked as missing.

In pandas, one of the most common ways that missing data is **introduced** into a data set is by reindexing. For example

```
In [1]: df = pd.DataFrame(np.random.randn(5, 3), index=['a', 'c', 'e', 'f', 'h'],
   ...:                   columns=['one', 'two', 'three'])
   ...:

In [2]: df['four'] = 'bar'

In [3]: df['five'] = df['one'] > 0

In [4]: df
Out[4]:
        one       two     three four   five
a  0.469112 -0.282863 -1.509059  bar   True
c -1.135632  1.212112 -0.173215  bar  False
e  0.119209 -1.044236 -0.861849  bar   True
f -2.104569 -0.494929  1.071804  bar  False
h  0.721555 -0.706771 -1.039575  bar   True

In [5]: df2 = df.reindex(['a', 'b', 'c', 'd', 'e', 'f', 'g', 'h'])

In [6]: df2
Out[6]:
        one       two     three four   five
a  0.469112 -0.282863 -1.509059  bar   True
b       NaN       NaN       NaN  NaN    NaN
c -1.135632  1.212112 -0.173215  bar  False
d       NaN       NaN       NaN  NaN    NaN
```

```
e  0.119209 -1.044236 -0.861849  bar   True
f -2.104569 -0.494929  1.071804  bar  False
g       NaN       NaN       NaN  NaN    NaN
h  0.721555 -0.706771 -1.039575  bar   True
```

Values considered "missing"

As data comes in many shapes and forms, pandas aims to be flexible with regard to handling missing data. While `NaN` is the default missing value marker for reasons of computational speed and convenience, we need to be able to easily detect this value with data of different types: floating point, integer, boolean, and general object. In many cases, however, the Python `None` will arise and we wish to also consider that "missing" or "null".

> **Note:** Prior to version v0.10.0 `inf` and `-inf` were also considered to be "null" in computations. This is no longer the case by default; use the `mode.use_inf_as_null` option to recover it.

To make detecting missing values easier (and across different array dtypes), pandas provides the **`isnull()`** and **`notnull()`** functions, which are also methods on `Series` and `DataFrame` objects:

```
In [7]: df2['one']
Out[7]:
a    0.469112
b         NaN
c   -1.135632
d         NaN
e    0.119209
f   -2.104569
g         NaN
h    0.721555
Name: one, dtype: float64

In [8]: pd.isnull(df2['one'])
Out[8]:
a    False
b     True
c    False
d     True
e    False
f    False
g     True
h    False
Name: one, dtype: bool

In [9]: df2['four'].notnull()
Out[9]:
a     True
b    False
c     True
d    False
e     True
f     True
g    False
h     True
Name: four, dtype: bool
```

```
In [10]: df2.isnull()
Out[10]:
     one    two  three   four   five
a  False  False  False  False  False
b   True   True   True   True   True
c  False  False  False  False  False
d   True   True   True   True   True
e  False  False  False  False  False
f  False  False  False  False  False
g   True   True   True   True   True
h  False  False  False  False  False
```

Warning: One has to be mindful that in python (and numpy), the `nan's` don't compare equal, but `None's` **do**. Note that Pandas/numpy uses the fact that `np.nan != np.nan`, and treats `None` like `np.nan`.

```
In [11]: None == None
Out[11]: True

In [12]: np.nan == np.nan
Out[12]: False
```

So as compared to above, a scalar equality comparison versus a `None/np.nan` doesn't provide useful information.

```
In [13]: df2['one'] == np.nan
Out[13]:
a    False
b    False
c    False
d    False
e    False
f    False
g    False
h    False
Name: one, dtype: bool
```

Datetimes

For datetime64[ns] types, `NaT` represents missing values. This is a pseudo-native sentinel value that can be represented by numpy in a singular dtype (datetime64[ns]). pandas objects provide intercompatibility between `NaT` and `NaN`.

```
In [14]: df2 = df.copy()

In [15]: df2['timestamp'] = pd.Timestamp('20120101')

In [16]: df2
Out[16]:
        one       two     three four   five  timestamp
a  0.469112 -0.282863 -1.509059  bar   True 2012-01-01
```

```
c -1.135632  1.212112 -0.173215  bar  False 2012-01-01
e  0.119209 -1.044236 -0.861849  bar   True 2012-01-01
f -2.104569 -0.494929  1.071804  bar  False 2012-01-01
h  0.721555 -0.706771 -1.039575  bar   True 2012-01-01

In [17]: df2.ix[['a','c','h'],['one','timestamp']] = np.nan

In [18]: df2
Out[18]:
        one       two     three four   five  timestamp
a       NaN -0.282863 -1.509059  bar   True        NaT
c       NaN  1.212112 -0.173215  bar  False        NaT
e  0.119209 -1.044236 -0.861849  bar   True 2012-01-01
f -2.104569 -0.494929  1.071804  bar  False 2012-01-01
h       NaN -0.706771 -1.039575  bar   True        NaT

In [19]: df2.get_dtype_counts()
Out[19]:
bool              1
datetime64[ns]    1
float64           3
object            1
dtype: int64
```

Inserting missing data

You can insert missing values by simply assigning to containers. The actual missing value used will be chosen based on the dtype.

For example, numeric containers will always use `NaN` regardless of the missing value type chosen:

```
In [20]: s = pd.Series([1, 2, 3])

In [21]: s.loc[0] = None

In [22]: s
Out[22]:
0   NaN
1     2
2     3
dtype: float64
```

Likewise, datetime containers will always use `NaT`.

For object containers, pandas will use the value given:

```
In [23]: s = pd.Series(["a", "b", "c"])

In [24]: s.loc[0] = None

In [25]: s.loc[1] = np.nan

In [26]: s
Out[26]:
0    None
```

```
1      NaN
2        c
dtype: object
```

Calculations with missing data

Missing values propagate naturally through arithmetic operations between pandas objects.

```
In [27]: a
Out[27]:
        one       two
a       NaN -0.282863
c       NaN  1.212112
e  0.119209 -1.044236
f -2.104569 -0.494929
h -2.104569 -0.706771

In [28]: b
Out[28]:
        one       two     three
a       NaN -0.282863 -1.509059
c       NaN  1.212112 -0.173215
e  0.119209 -1.044236 -0.861849
f -2.104569 -0.494929  1.071804
h       NaN -0.706771 -1.039575

In [29]: a + b
Out[29]:
        one  three       two
a       NaN    NaN -0.565727
c       NaN    NaN  2.424224
e  0.238417    NaN -2.088472
f -4.209138    NaN -0.989859
h       NaN    NaN -1.413542
```

The descriptive statistics and computational methods discussed in the *data structure overview* (and listed *here* and *here*) are all written to account for missing data. For example:

- When summing data, NA (missing) values will be treated as zero
- If the data are all NA, the result will be NA
- Methods like **cumsum** and **cumprod** ignore NA values, but preserve them in the resulting arrays

```
In [30]: df
Out[30]:
        one       two     three
a       NaN -0.282863 -1.509059
c       NaN  1.212112 -0.173215
e  0.119209 -1.044236 -0.861849
f -2.104569 -0.494929  1.071804
h       NaN -0.706771 -1.039575

In [31]: df['one'].sum()
Out[31]: -1.9853605075978744

In [32]: df.mean(1)
```

```
Out[32]:
a   -0.895961
c    0.519449
e   -0.595625
f   -0.509232
h   -0.873173
dtype: float64

In [33]: df.cumsum()
Out[33]:
        one       two     three
a       NaN -0.282863 -1.509059
c       NaN  0.929249 -1.682273
e  0.119209 -0.114987 -2.544122
f -1.985361 -0.609917 -1.472318
h       NaN -1.316688 -2.511893
```

NA values in GroupBy

NA groups in GroupBy are automatically excluded. This behavior is consistent with R, for example:

```
In [34]: df
Out[34]:
        one       two     three
a       NaN -0.282863 -1.509059
c       NaN  1.212112 -0.173215
e  0.119209 -1.044236 -0.861849
f -2.104569 -0.494929  1.071804
h       NaN -0.706771 -1.039575

In [35]: df.groupby('one').mean()
Out[35]:
                two     three
one
-2.104569 -0.494929  1.071804
 0.119209 -1.044236 -0.861849
```

See the groupby section *here* for more information.

Cleaning / filling missing data

pandas objects are equipped with various data manipulation methods for dealing with missing data.

Filling missing values: fillna

The **fillna** function can “fill in” NA values with non-null data in a couple of ways, which we illustrate:

Replace NA with a scalar value

```
In [36]: df2
Out[36]:
        one       two     three four   five  timestamp
a       NaN -0.282863 -1.509059  bar   True        NaT
c       NaN  1.212112 -0.173215  bar  False        NaT
e  0.119209 -1.044236 -0.861849  bar   True 2012-01-01
f -2.104569 -0.494929  1.071804  bar  False 2012-01-01
h       NaN -0.706771 -1.039575  bar   True        NaT

In [37]: df2.fillna(0)
Out[37]:
        one       two     three four   five  timestamp
a  0.000000 -0.282863 -1.509059  bar   True 1970-01-01
c  0.000000  1.212112 -0.173215  bar  False 1970-01-01
e  0.119209 -1.044236 -0.861849  bar   True 2012-01-01
f -2.104569 -0.494929  1.071804  bar  False 2012-01-01
h  0.000000 -0.706771 -1.039575  bar   True 1970-01-01

In [38]: df2['four'].fillna('missing')
Out[38]:
a    bar
c    bar
e    bar
f    bar
h    bar
Name: four, dtype: object
```

Fill gaps forward or backward

Using the same filling arguments as *reindexing*, we can propagate non-null values forward or backward:

```
In [39]: df
Out[39]:
        one       two     three
a       NaN -0.282863 -1.509059
c       NaN  1.212112 -0.173215
e  0.119209 -1.044236 -0.861849
f -2.104569 -0.494929  1.071804
h       NaN -0.706771 -1.039575

In [40]: df.fillna(method='pad')
Out[40]:
        one       two     three
a       NaN -0.282863 -1.509059
c       NaN  1.212112 -0.173215
e  0.119209 -1.044236 -0.861849
f -2.104569 -0.494929  1.071804
h -2.104569 -0.706771 -1.039575
```

Limit the amount of filling

If we only want consecutive gaps filled up to a certain number of data points, we can use the *limit* keyword:

```
In [41]: df
Out[41]:
   one       two     three
```

```
a  NaN -0.282863 -1.509059
c  NaN  1.212112 -0.173215
e  NaN       NaN       NaN
f  NaN       NaN       NaN
h  NaN -0.706771 -1.039575

In [42]: df.fillna(method='pad', limit=1)
Out[42]: 
   one       two     three
a  NaN -0.282863 -1.509059
c  NaN  1.212112 -0.173215
e  NaN  1.212112 -0.173215
f  NaN       NaN       NaN
h  NaN -0.706771 -1.039575
```

To remind you, these are the available filling methods:

Method	Action
pad / ffill	Fill values forward
bfill / backfill	Fill values backward

With time series data, using pad/ffill is extremely common so that the “last known value” is available at every time point.

The `ffill()` function is equivalent to `fillna(method='ffill')` and `bfill()` is equivalent to `fillna(method='bfill')`

Filling with a PandasObject

New in version 0.12.

You can also fillna using a dict or Series that is alignable. The labels of the dict or index of the Series must match the columns of the frame you wish to fill. The use case of this is to fill a DataFrame with the mean of that column.

```
In [43]: dff = pd.DataFrame(np.random.randn(10,3), columns=list('ABC'))

In [44]: dff.iloc[3:5,0] = np.nan

In [45]: dff.iloc[4:6,1] = np.nan

In [46]: dff.iloc[5:8,2] = np.nan

In [47]: dff
Out[47]: 
          A         B         C
0  0.271860 -0.424972  0.567020
1  0.276232 -1.087401 -0.673690
2  0.113648 -1.478427  0.524988
3       NaN  0.577046 -1.715002
4       NaN       NaN -1.157892
5 -1.344312       NaN       NaN
6 -0.109050  1.643563       NaN
7  0.357021 -0.674600       NaN
8 -0.968914 -1.294524  0.413738
9  0.276662 -0.472035 -0.013960
```

```
In [48]: dff.fillna(dff.mean())
Out[48]: 
          A         B         C
0  0.271860 -0.424972  0.567020
1  0.276232 -1.087401 -0.673690
2  0.113648 -1.478427  0.524988
3 -0.140857  0.577046 -1.715002
4 -0.140857 -0.401419 -1.157892
5 -1.344312 -0.401419 -0.293543
6 -0.109050  1.643563 -0.293543
7  0.357021 -0.674600 -0.293543
8 -0.968914 -1.294524  0.413738
9  0.276662 -0.472035 -0.013960

In [49]: dff.fillna(dff.mean()['B':'C'])
Out[49]: 
          A         B         C
0  0.271860 -0.424972  0.567020
1  0.276232 -1.087401 -0.673690
2  0.113648 -1.478427  0.524988
3       NaN  0.577046 -1.715002
4       NaN -0.401419 -1.157892
5 -1.344312 -0.401419 -0.293543
6 -0.109050  1.643563 -0.293543
7  0.357021 -0.674600 -0.293543
8 -0.968914 -1.294524  0.413738
9  0.276662 -0.472035 -0.013960
```

New in version 0.13.

Same result as above, but is aligning the 'fill' value which is a Series in this case.

```
In [50]: dff.where(pd.notnull(dff), dff.mean(), axis='columns')
Out[50]: 
          A         B         C
0  0.271860 -0.424972  0.567020
1  0.276232 -1.087401 -0.673690
2  0.113648 -1.478427  0.524988
3 -0.140857  0.577046 -1.715002
4 -0.140857 -0.401419 -1.157892
5 -1.344312 -0.401419 -0.293543
6 -0.109050  1.643563 -0.293543
7  0.357021 -0.674600 -0.293543
8 -0.968914 -1.294524  0.413738
9  0.276662 -0.472035 -0.013960
```

Dropping axis labels with missing data: dropna

You may wish to simply exclude labels from a data set which refer to missing data. To do this, use the **dropna** method:

```
In [51]: df
Out[51]: 
   one       two     three
a  NaN -0.282863 -1.509059
c  NaN  1.212112 -0.173215
e  NaN  0.000000  0.000000
```

```
f  NaN  0.000000  0.000000
h  NaN -0.706771 -1.039575

In [52]: df.dropna(axis=0)
Out[52]:
Empty DataFrame
Columns: [one, two, three]
Index: []

In [53]: df.dropna(axis=1)
Out[53]:
        two     three
a -0.282863 -1.509059
c  1.212112 -0.173215
e  0.000000  0.000000
f  0.000000  0.000000
h -0.706771 -1.039575

In [54]: df['one'].dropna()
Out[54]: Series([], Name: one, dtype: float64)
```

Series.dropna is a simpler method as it only has one axis to consider. DataFrame.dropna has considerably more options than Series.dropna, which can be examined *in the API*.

Interpolation

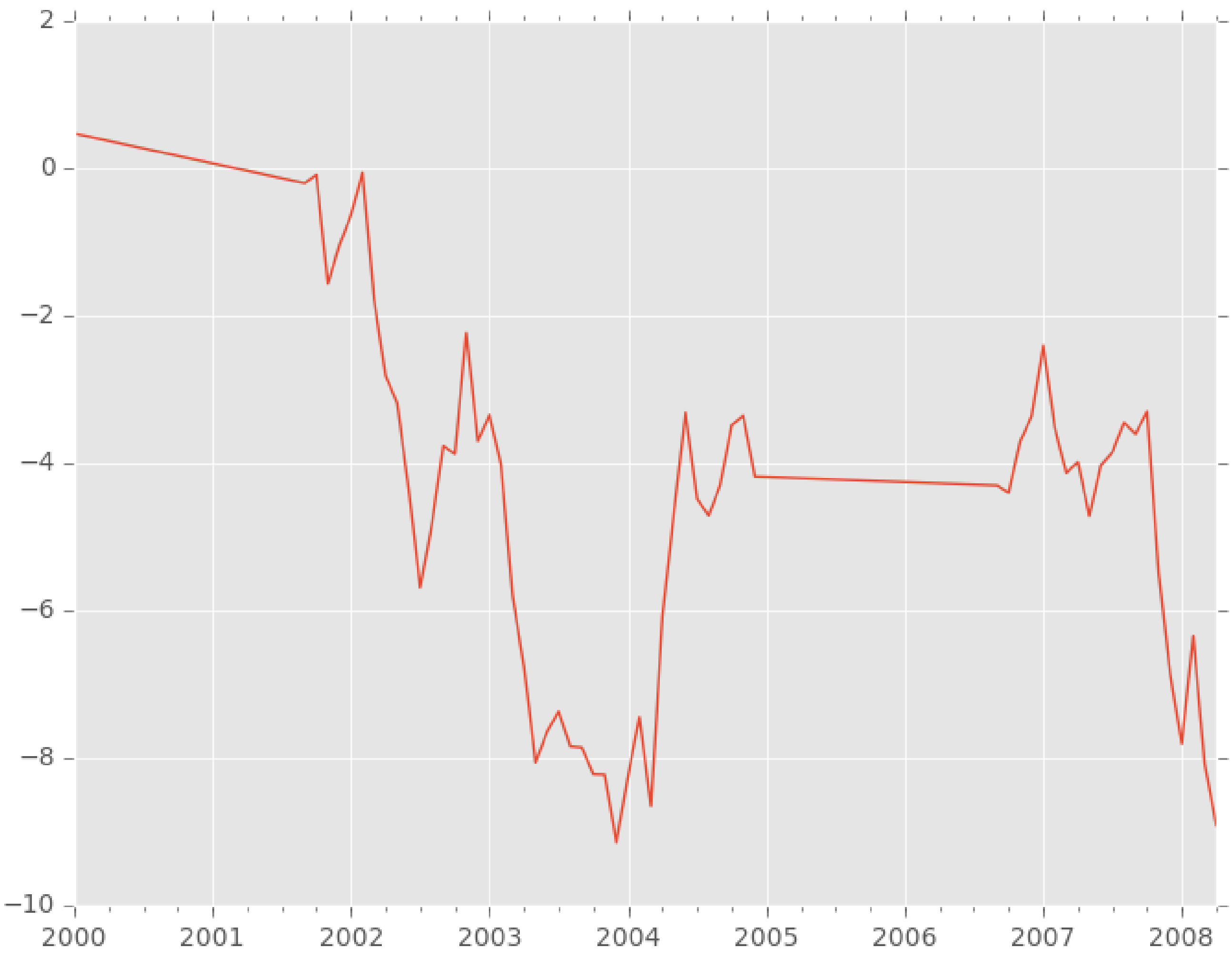
New in version 0.13.0: **`interpolate()`**, and **`interpolate()`** have revamped interpolation methods and functionality.

New in version 0.17.0: The `limit_direction` keyword argument was added.

Both Series and Dataframe objects have an `interpolate` method that, by default, performs linear interpolation at missing datapoints.

```
In [55]: ts
Out[55]:
2000-01-31    0.469112
2000-02-29         NaN
2000-03-31         NaN
2000-04-28         NaN
2000-05-31         NaN
2000-06-30         NaN
2000-07-31         NaN
                 ...
2007-10-31   -3.305259
2007-11-30   -5.485119
2007-12-31   -6.854968
2008-01-31   -7.809176
2008-02-29   -6.346480
2008-03-31   -8.089641
2008-04-30   -8.916232
Freq: BM, dtype: float64

In [56]: ts.count()
Out[56]: 61

In [57]: ts.interpolate().count()
Out[57]: 100
```

```
In [58]: ts.interpolate().plot()
Out[58]: <matplotlib.axes._subplots.AxesSubplot at 0x9c3ec48c>
```



Index aware interpolation is available via the `method` keyword:

```
In [59]: ts2
Out[59]:
2000-01-31    0.469112
2000-02-29         NaN
2002-07-31   -5.689738
2005-01-31         NaN
2008-04-30   -8.916232
dtype: float64

In [60]: ts2.interpolate()
Out[60]:
2000-01-31    0.469112
2000-02-29   -2.610313
2002-07-31   -5.689738
2005-01-31   -7.302985
2008-04-30   -8.916232
dtype: float64

In [61]: ts2.interpolate(method='time')
Out[61]:
2000-01-31    0.469112
2000-02-29    0.273272
2002-07-31   -5.689738
```

```
2005-01-31   -7.095568
2008-04-30   -8.916232
dtype: float64
```

For a floating-point index, use `method='values'`:

```
In [62]: ser
Out[62]:
0      0
1    NaN
10    10
dtype: float64

In [63]: ser.interpolate()
Out[63]:
0      0
1      5
10    10
dtype: float64

In [64]: ser.interpolate(method='values')
Out[64]:
0      0
1      1
10    10
dtype: float64
```

You can also interpolate with a DataFrame:

```
In [65]: df = pd.DataFrame({'A': [1, 2.1, np.nan, 4.7, 5.6, 6.8],
   ....:                    'B': [.25, np.nan, np.nan, 4, 12.2, 14.4]})
   ....:

In [66]: df
Out[66]:
     A      B
0  1.0   0.25
1  2.1    NaN
2  NaN    NaN
3  4.7   4.00
4  5.6  12.20
5  6.8  14.40

In [67]: df.interpolate()
Out[67]:
     A      B
0  1.0   0.25
1  2.1   1.50
2  3.4   2.75
3  4.7   4.00
4  5.6  12.20
5  6.8  14.40
```

The `method` argument gives access to fancier interpolation methods. If you have scipy installed, you can set pass the name of a 1-d interpolation routine to `method`. You'll want to consult the full scipy interpolation documentation and reference guide for details. The appropriate interpolation method will depend on the type of data you are working with. For example, if you are dealing with a time series that is growing at an increasing rate, `method='quadratic'` may be appropriate. If you have values approximating a cumulative distribution function, then `method='pchip'` should work well.

Warning: These methods require `scipy`.

```
In [68]: df.interpolate(method='barycentric')
Out[68]:
      A       B
0  1.00   0.250
1  2.10  -7.660
2  3.53  -4.515
3  4.70   4.000
4  5.60  12.200
5  6.80  14.400

In [69]: df.interpolate(method='pchip')
Out[69]:
          A          B
0  1.000000   0.250000
1  2.100000   1.130135
2  3.429309   2.337586
3  4.700000   4.000000
4  5.600000  12.200000
5  6.800000  14.400000
```

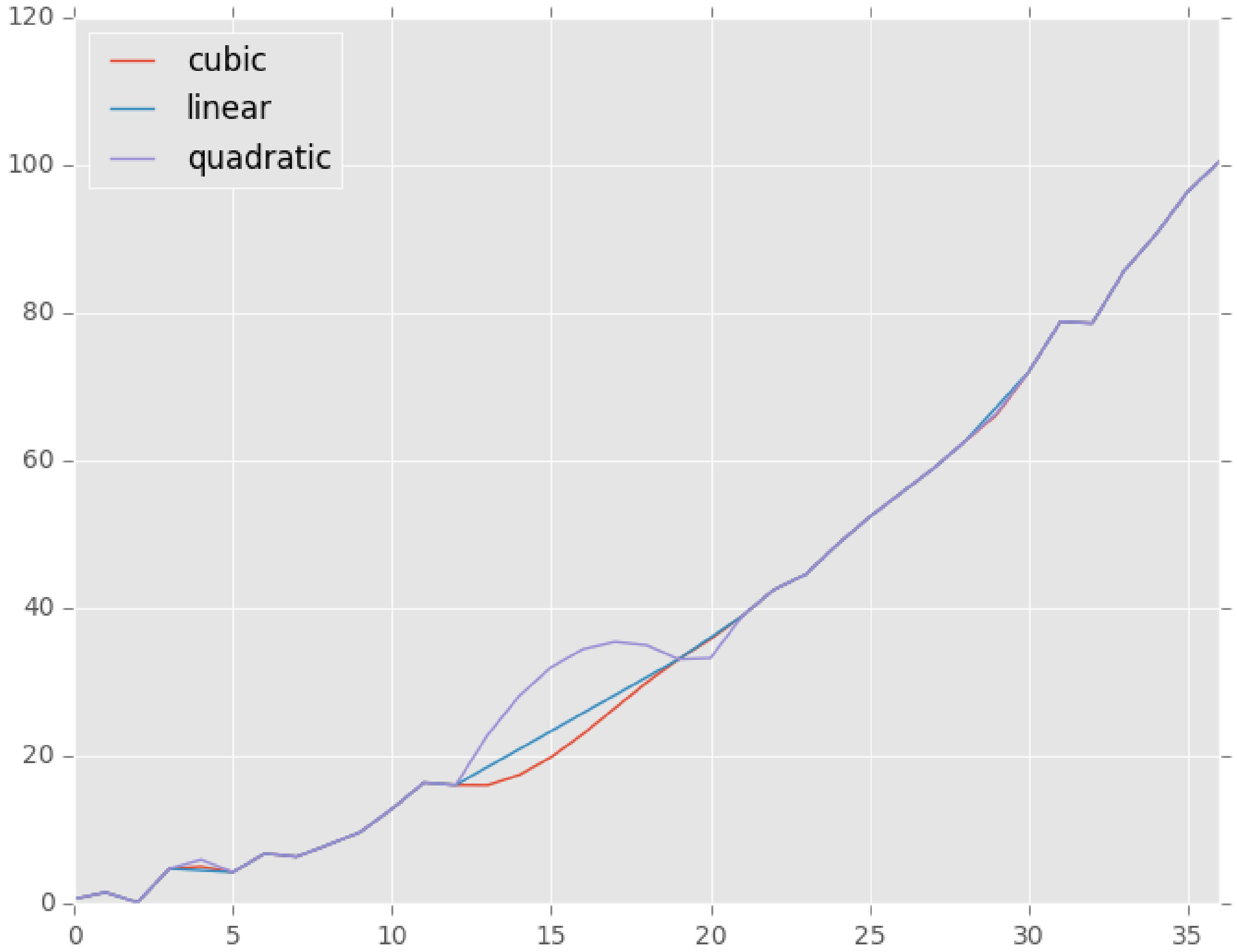
When interpolating via a polynomial or spline approximation, you must also specify the degree or order of the approximation:

```
In [70]: df.interpolate(method='spline', order=2)
Out[70]:
          A          B
0  1.000000   0.250000
1  2.100000  -0.428598
2  3.404545   1.206900
3  4.700000   4.000000
4  5.600000  12.200000
5  6.800000  14.400000

In [71]: df.interpolate(method='polynomial', order=2)
Out[71]:
          A          B
0  1.000000   0.250000
1  2.100000  -4.161538
2  3.547059  -2.911538
3  4.700000   4.000000
4  5.600000  12.200000
5  6.800000  14.400000
```

Compare several methods:

```
In [72]: np.random.seed(2)

In [73]: ser = pd.Series(np.arange(1, 10.1, .25)**2 + np.random.randn(37))

In [74]: bad = np.array([4, 13, 14, 15, 16, 17, 18, 20, 29])
```

```
In [75]: ser[bad] = np.nan

In [76]: methods = ['linear', 'quadratic', 'cubic']

In [77]: df = pd.DataFrame({m: ser.interpolate(method=m) for m in methods})

In [78]: df.plot()
Out[78]: <matplotlib.axes._subplots.AxesSubplot at 0x9c36626c>
```



Another use case is interpolation at *new* values. Suppose you have 100 observations from some distribution. And let's suppose that you're particularly interested in what's happening around the middle. You can mix pandas' `reindex` and `interpolate` methods to interpolate at the new values.

```
In [79]: ser = pd.Series(np.sort(np.random.uniform(size=100)))

# interpolate at new_index
In [80]: new_index = ser.index | pd.Index([49.25, 49.5, 49.75, 50.25, 50.5, 50.75])

In [81]: interp_s = ser.reindex(new_index).interpolate(method='pchip')

In [82]: interp_s[49:51]
Out[82]:
49.00    0.471410
49.25    0.476841
49.50    0.481780
49.75    0.485998
50.00    0.489266
```

```
50.25    0.491814
50.50    0.493995
50.75    0.495763
51.00    0.497074
dtype: float64
```

Interpolation Limits

Like other pandas fill methods, `interpolate` accepts a `limit` keyword argument. Use this argument to limit the number of consecutive interpolations, keeping `NaN` values for interpolations that are too far from the last valid observation:

```
In [83]: ser = pd.Series([np.nan, np.nan, 5, np.nan, np.nan, np.nan, 13])

In [84]: ser.interpolate(limit=2)
Out[84]:
0    NaN
1    NaN
2      5
3      7
4      9
5    NaN
6     13
dtype: float64
```

By default, `limit` applies in a forward direction, so that only `NaN` values after a non-`NaN` value can be filled. If you provide `'backward'` or `'both'` for the `limit_direction` keyword argument, you can fill `NaN` values before non-`NaN` values, or both before and after non-`NaN` values, respectively:

```
In [85]: ser.interpolate(limit=1)  # limit_direction == 'forward'
Out[85]:
0    NaN
1    NaN
2      5
3      7
4    NaN
5    NaN
6     13
dtype: float64

In [86]: ser.interpolate(limit=1, limit_direction='backward')
Out[86]:
0    NaN
1      5
2      5
3    NaN
4    NaN
5     11
6     13
dtype: float64

In [87]: ser.interpolate(limit=1, limit_direction='both')
Out[87]:
0    NaN
1      5
2      5
3      7
```

```
4    NaN
5     11
6     13
dtype: float64
```

Replacing Generic Values

Often times we want to replace arbitrary values with other values. New in v0.8 is the `replace` method in Series/DataFrame that provides an efficient yet flexible way to perform such replacements.

For a Series, you can replace a single value or a list of values by another value:

```
In [88]: ser = pd.Series([0., 1., 2., 3., 4.])

In [89]: ser.replace(0, 5)
Out[89]:
0    5
1    1
2    2
3    3
4    4
dtype: float64
```

You can replace a list of values by a list of other values:

```
In [90]: ser.replace([0, 1, 2, 3, 4], [4, 3, 2, 1, 0])
Out[90]:
0    4
1    3
2    2
3    1
4    0
dtype: float64
```

You can also specify a mapping dict:

```
In [91]: ser.replace({0: 10, 1: 100})
Out[91]:
0     10
1    100
2      2
3      3
4      4
dtype: float64
```

For a DataFrame, you can specify individual values by column:

```
In [92]: df = pd.DataFrame({'a': [0, 1, 2, 3, 4], 'b': [5, 6, 7, 8, 9]})

In [93]: df.replace({'a': 0, 'b': 5}, 100)
Out[93]:
```

```
     a    b
0  100  100
1    1    6
2    2    7
3    3    8
4    4    9
```

Instead of replacing with specified values, you can treat all given values as missing and interpolate over them:

```
In [94]: ser.replace([1, 2, 3], method='pad')
Out[94]:
0    0
1    0
2    0
3    0
4    4
dtype: float64
```

String/Regular Expression Replacement

Note: Python strings prefixed with the `r` character such as `r'hello world'` are so-called "raw" strings. They have different semantics regarding backslashes than strings without this prefix. Backslashes in raw strings will be interpreted as an escaped backslash, e.g., `r'\' == '\\'`. You should read about them if this is unclear.

Replace the '.' with `nan` (str -> str)

```
In [95]: d = {'a': list(range(4)), 'b': list('ab..'), 'c': ['a', 'b', np.nan, 'd']}

In [96]: df = pd.DataFrame(d)

In [97]: df.replace('.', np.nan)
Out[97]:
   a    b    c
0  0    a    a
1  1    b    b
2  2  NaN  NaN
3  3  NaN    d
```

Now do it with a regular expression that removes surrounding whitespace (regex -> regex)

```
In [98]: df.replace(r'\s*\.\s*', np.nan, regex=True)
Out[98]:
   a    b    c
0  0    a    a
1  1    b    b
2  2  NaN  NaN
3  3  NaN    d
```

Replace a few different values (list -> list)

```
In [99]: df.replace(['a', '.'], ['b', np.nan])
Out[99]:
   a    b    c
0  0    b    b
1  1    b    b
2  2  NaN  NaN
3  3  NaN    d
```

list of regex -> list of regex

```
In [100]: df.replace([r'\.', r'(a)'], ['dot', '\1stuff'], regex=True)
Out[100]:
   a      b      c
0  0  stuff  stuff
1  1      b      b
2  2    dot    NaN
3  3    dot      d
```

Only search in column `'b'` (dict -> dict)

```
In [101]: df.replace({'b': '.'}, {'b': np.nan})
Out[101]:
   a    b    c
0  0    a    a
1  1    b    b
2  2  NaN  NaN
3  3  NaN    d
```

Same as the previous example, but use a regular expression for searching instead (dict of regex -> dict)

```
In [102]: df.replace({'b': r'\s*\.\s*'}, {'b': np.nan}, regex=True)
Out[102]:
   a    b    c
0  0    a    a
1  1    b    b
2  2  NaN  NaN
3  3  NaN    d
```

You can pass nested dictionaries of regular expressions that use `regex=True`

```
In [103]: df.replace({'b': {'b': r''}}, regex=True)
Out[103]:
   a  b    c
0  0  a    a
1  1       b
2  2  .  NaN
3  3  .    d
```

or you can pass the nested dictionary like so

```
In [104]: df.replace(regex={'b': {r'\s*\.\s*': np.nan}})
```

```
Out[104]:
   a    b    c
0  0    a    a
1  1    b    b
2  2  NaN  NaN
3  3  NaN    d
```

You can also use the group of a regular expression match when replacing (dict of regex -> dict of regex), this works for lists as well

```
In [105]: df.replace({'b': r'\s*(\.)\s*'}, {'b': r'\1ty'}, regex=True)
Out[105]:
   a    b    c
0  0    a    a
1  1    b    b
2  2  .ty  NaN
3  3  .ty    d
```

You can pass a list of regular expressions, of which those that match will be replaced with a scalar (list of regex -> regex)

```
In [106]: df.replace([r'\s*\.\s*', r'a|b'], np.nan, regex=True)
Out[106]:
   a   b    c
0  0 NaN  NaN
1  1 NaN  NaN
2  2 NaN  NaN
3  3 NaN    d
```

All of the regular expression examples can also be passed with the `to_replace` argument as the `regex` argument. In this case the `value` argument must be passed explicitly by name or `regex` must be a nested dictionary. The previous example, in this case, would then be

```
In [107]: df.replace(regex=[r'\s*\.\s*', r'a|b'], value=np.nan)
Out[107]:
   a   b    c
0  0 NaN  NaN
1  1 NaN  NaN
2  2 NaN  NaN
3  3 NaN    d
```

This can be convenient if you do not want to pass `regex=True` every time you want to use a regular expression.

Note: Anywhere in the above `replace` examples that you see a regular expression a compiled regular expression is valid as well.

Numeric Replacement

Similar to `DataFrame.fillna`

```
In [108]: df = pd.DataFrame(np.random.randn(10, 2))

In [109]: df[np.random.rand(df.shape[0]) > 0.5] = 1.5

In [110]: df.replace(1.5, np.nan)
Out[110]:
          0         1
0 -0.844214 -1.021415
1  0.432396 -0.323580
2  0.423825  0.799180
3  1.262614  0.751965
4       NaN       NaN
5       NaN       NaN
6 -0.498174 -1.060799
7  0.591667 -0.183257
8  1.019855 -1.482465
9       NaN       NaN
```

Replacing more than one value via lists works as well

```
In [111]: df00 = df.values[0, 0]

In [112]: df.replace([1.5, df00], [np.nan, 'a'])
Out[112]:
          0         1
0         a -1.021415
1  0.432396 -0.323580
2  0.423825  0.799180
3   1.26261  0.751965
4       NaN       NaN
5       NaN       NaN
6 -0.498174 -1.060799
7  0.591667 -0.183257
8   1.01985 -1.482465
9       NaN       NaN

In [113]: df[1].dtype
Out[113]: dtype('float64')
```

You can also operate on the DataFrame in place

```
In [114]: df.replace(1.5, np.nan, inplace=True)
```

Warning: When replacing multiple `bool` or `datetime64` objects, the first argument to `replace` (`to_replace`) must match the type of the value being replaced type. For example,

```
s = pd.Series([True, False, True])
s.replace({'a string': 'new value', True: False})  # raises

TypeError: Cannot compare types 'ndarray(dtype=bool)' and 'str'
```

will raise a `TypeError` because one of the `dict` keys is not of the correct type for replacement.

However, when replacing a *single* object such as,

```
In [115]: s = pd.Series([True, False, True])

In [116]: s.replace('a string', 'another string')
Out[116]:
0     True
1    False
2     True
dtype: bool
```

the original `NDFrame` object will be returned untouched. We're working on unifying this API, but for backwards compatibility reasons we cannot break the latter behavior. See GH6354 for more details.

Missing data casting rules and indexing

While pandas supports storing arrays of integer and boolean type, these types are not capable of storing missing data. Until we can switch to using a native NA type in NumPy, we've established some "casting rules" when reindexing will cause missing data to be introduced into, say, a Series or DataFrame. Here they are:

data type	Cast to
integer	float
boolean	object
float	no cast
object	no cast

For example:

```
In [117]: s = pd.Series(np.random.randn(5), index=[0, 2, 4, 6, 7])

In [118]: s > 0
Out[118]:
0    True
2    True
4    True
6    True
7    True
dtype: bool

In [119]: (s > 0).dtype
Out[119]: dtype('bool')

In [120]: crit = (s > 0).reindex(list(range(8)))

In [121]: crit
Out[121]:
0    True
1     NaN
2    True
3     NaN
4    True
5     NaN
6    True
7    True
```

```
dtype: object

In [122]: crit.dtype
Out[122]: dtype('O')
```

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

```
In [123]: reindexed = s.reindex(list(range(8))).fillna(0)

In [124]: reindexed[crit]
---------------------------------------------------------------------------
ValueError                                Traceback (most recent call last)
<ipython-input-124-2da204ed1ac7> in <module>()
----> 1 reindexed[crit]

/home/joris/scipy/pandas/pandas/core/series.pyc in __getitem__(self, key)
    592             key = list(key)
    593 
--> 594         if is_bool_indexer(key):
    595             key = check_bool_indexer(self.index, key)
    596 

/home/joris/scipy/pandas/pandas/core/common.pyc in is_bool_indexer(key)
   1737             if not lib.is_bool_array(key):
   1738                 if isnull(key).any():
-> 1739                     raise ValueError('cannot index with vector containing '
   1740                                      'NA / NaN values')
   1741                 return False

ValueError: cannot index with vector containing NA / NaN values
```

However, these can be filled in using **fillna** and it will work fine:

```
In [125]: reindexed[crit.fillna(False)]
Out[125]: 
0    0.126504
2    0.696198
4    0.697416
6    0.601516
7    0.003659
dtype: float64

In [126]: reindexed[crit.fillna(True)]
Out[126]: 
0    0.126504
1    0.000000
2    0.696198
3    0.000000
4    0.697416
5    0.000000
6    0.601516
7    0.003659
dtype: float64
```