Why not make NumPy like R?

Many people have suggested that NumPy should simply emulate the NA support present in the more domain-specific statistical programming language R. Part of the reason is the NumPy type hierarchy:

Typeclass	Dtypes
numpy.floating	float16, float32, float64, float128
numpy.integer	int8, int16, int32, int64
numpy.unsignedinteger	uint8, uint16, uint32, uint64
numpy.object_	object_
numpy.bool_	bool_
numpy.character	string_, unicode_

The R language, by contrast, only has a handful of built-in data types: integer, numeric (floating-point), character, and boolean. NA types are implemented by reserving special bit patterns for each type to be used as the missing value. While doing this with the full NumPy type hierarchy would be possible, it would be a more substantial trade-off (especially for the 8- and 16-bit data types) and implementation undertaking.

An alternate approach is that of using masked arrays. A masked array is an array of data with an associated boolean *mask* denoting whether each value should be considered NA or not. I am personally not in love with this approach as I feel that overall it places a fairly heavy burden on the user and the library implementer. Additionally, it exacts a fairly high performance cost when working with numerical data compared with the simple approach of using NaN. Thus, I have chosen the Pythonic "practicality beats purity" approach and traded integer NA capability for a much simpler approach of using a special value in float and object arrays to denote NA, and promoting integer arrays to floating when NAs must be introduced.

2.21.4 Differences with NumPy

For Series and DataFrame objects, *var()* normalizes by N-1 to produce unbiased estimates of the sample variance, while NumPy's var normalizes by N, which measures the variance of the sample. Note that *cov()* normalizes by N-1 in both pandas and NumPy.

2.21.5 Thread-safety

As of pandas 0.11, pandas is not 100% thread safe. The known issues relate to the *copy()* method. If you are doing a lot of copying of DataFrame objects shared among threads, we recommend holding locks inside the threads where the data copying occurs.

See this link for more information.

2.21.6 Byte-Ordering issues

Occasionally you may have to deal with data that were created on a machine with a different byte order than the one on which you are running Python. A common symptom of this issue is an error like::

```
Traceback
    ...
ValueError: Big-endian buffer not supported on little-endian compiler
```

To deal with this issue you should convert the underlying NumPy array to the native system byte order *before* passing it to Series or DataFrame constructors using something similar to the following:

```
In [32]: x = np.array(list(range(10)), '>i4')  # big endian

In [33]: newx = x.byteswap().newbyteorder()  # force native byteorder

In [34]: s = pd.Series(newx)
```

See the NumPy documentation on byte order for more details.

2.22 Cookbook

This is a repository for *short and sweet* examples and links for useful pandas recipes. We encourage users to add to this documentation.

Adding interesting links and/or inline examples to this section is a great *First Pull Request*.

Simplified, condensed, new-user friendly, in-line examples have been inserted where possible to augment the Stack-Overflow and GitHub links. Many of the links contain expanded information, above what the in-line examples offer.

Pandas (pd) and Numpy (np) are the only two abbreviated imported modules. The rest are kept explicitly imported for newer users.

These examples are written for Python 3. Minor tweaks might be necessary for earlier python versions.

2.22.1 Idioms

These are some neat pandas `idioms`

if-then/if-then-else on one column, and assignment to another one or more columns:

```
In [1]: df = pd.DataFrame({'AAA': [4, 5, 6, 7],
   ...:                    'BBB': [10, 20, 30, 40],
   ...:                    'CCC': [100, 50, -30, -50]})
   ...:

In [2]: df
Out[2]:
   AAA  BBB  CCC
0    4   10  100
1    5   20   50
2    6   30  -30
3    7   40  -50
```

if-then...

An if-then on one column

```
In [3]: df.loc[df.AAA >= 5, 'BBB'] = -1

In [4]: df
Out[4]:
   AAA  BBB  CCC
0    4   10  100
1    5   -1   50
```

(continues on next page)

(continued from previous page)

```
2    6   -1  -30
3    7   -1  -50
```

An if-then with assignment to 2 columns:

```
In [5]: df.loc[df.AAA >= 5, ['BBB', 'CCC']] = 555

In [6]: df
Out[6]:
   AAA  BBB  CCC
0    4   10  100
1    5  555  555
2    6  555  555
3    7  555  555
```

Add another line with different logic, to do the -else

```
In [7]: df.loc[df.AAA < 5, ['BBB', 'CCC']] = 2000

In [8]: df
Out[8]:
   AAA   BBB   CCC
0    4  2000  2000
1    5   555   555
2    6   555   555
3    7   555   555
```

Or use pandas where after you've set up a mask

```
In [9]: df_mask = pd.DataFrame({'AAA': [True] * 4,
   ...:                         'BBB': [False] * 4,
   ...:                         'CCC': [True, False] * 2})
   ...:

In [10]: df.where(df_mask, -1000)
Out[10]:
   AAA   BBB   CCC
0    4 -1000  2000
1    5 -1000 -1000
2    6 -1000   555
3    7 -1000 -1000
```

if-then-else using numpy's where()

```
In [11]: df = pd.DataFrame({'AAA': [4, 5, 6, 7],
   ....:                    'BBB': [10, 20, 30, 40],
   ....:                    'CCC': [100, 50, -30, -50]})
   ....:

In [12]: df
Out[12]:
   AAA  BBB  CCC
0    4   10  100
1    5   20   50
2    6   30  -30
3    7   40  -50
```

(continues on next page)

(continued from previous page)

```
In [13]: df['logic'] = np.where(df['AAA'] > 5, 'high', 'low')

In [14]: df
Out[14]:
   AAA  BBB  CCC logic
0    4   10  100   low
1    5   20   50   low
2    6   30  -30  high
3    7   40  -50  high
```

Splitting

Split a frame with a boolean criterion

```
In [15]: df = pd.DataFrame({'AAA': [4, 5, 6, 7],
   ....:                    'BBB': [10, 20, 30, 40],
   ....:                    'CCC': [100, 50, -30, -50]})
   ....:

In [16]: df
Out[16]:
   AAA  BBB  CCC
0    4   10  100
1    5   20   50
2    6   30  -30
3    7   40  -50

In [17]: df[df.AAA <= 5]
Out[17]:
   AAA  BBB  CCC
0    4   10  100
1    5   20   50

In [18]: df[df.AAA > 5]
Out[18]:
   AAA  BBB  CCC
2    6   30  -30
3    7   40  -50
```

Building criteria

Select with multi-column criteria

```
In [19]: df = pd.DataFrame({'AAA': [4, 5, 6, 7],
   ....:                    'BBB': [10, 20, 30, 40],
   ....:                    'CCC': [100, 50, -30, -50]})
   ....:

In [20]: df
Out[20]:
   AAA  BBB  CCC
0    4   10  100
1    5   20   50
```

(continues on next page)

(continued from previous page)

```
2    6   30  -30
3    7   40  -50
```

…and (without assignment returns a Series)

```
In [21]: df.loc[(df['BBB'] < 25) & (df['CCC'] >= -40), 'AAA']
Out[21]:
0    4
1    5
Name: AAA, dtype: int64
```

…or (without assignment returns a Series)

```
In [22]: df.loc[(df['BBB'] > 25) | (df['CCC'] >= -40), 'AAA']
Out[22]:
0    4
1    5
2    6
3    7
Name: AAA, dtype: int64
```

…or (with assignment modifies the DataFrame.)

```
In [23]: df.loc[(df['BBB'] > 25) | (df['CCC'] >= 75), 'AAA'] = 0.1

In [24]: df
Out[24]:
   AAA  BBB  CCC
0  0.1   10  100
1  5.0   20   50
2  0.1   30  -30
3  0.1   40  -50
```

Select rows with data closest to certain value using argsort

```
In [25]: df = pd.DataFrame({'AAA': [4, 5, 6, 7],
   ....:                    'BBB': [10, 20, 30, 40],
   ....:                    'CCC': [100, 50, -30, -50]})
   ....:

In [26]: df
Out[26]:
   AAA  BBB  CCC
0    4   10  100
1    5   20   50
2    6   30  -30
3    7   40  -50

In [27]: aValue = 43.0

In [28]: df.loc[(df.CCC - aValue).abs().argsort()]
Out[28]:
   AAA  BBB  CCC
1    5   20   50
0    4   10  100
2    6   30  -30
3    7   40  -50
```

Dynamically reduce a list of criteria using a binary operators

```
In [29]: df = pd.DataFrame({'AAA': [4, 5, 6, 7],
   ....:                    'BBB': [10, 20, 30, 40],
   ....:                    'CCC': [100, 50, -30, -50]})
   ....:

In [30]: df
Out[30]:
   AAA  BBB  CCC
0    4   10  100
1    5   20   50
2    6   30  -30
3    7   40  -50

In [31]: Crit1 = df.AAA <= 5.5

In [32]: Crit2 = df.BBB == 10.0

In [33]: Crit3 = df.CCC > -40.0
```

One could hard code:

```
In [34]: AllCrit = Crit1 & Crit2 & Crit3
```

...Or it can be done with a list of dynamically built criteria

```
In [35]: import functools

In [36]: CritList = [Crit1, Crit2, Crit3]

In [37]: AllCrit = functools.reduce(lambda x, y: x & y, CritList)

In [38]: df[AllCrit]
Out[38]:
   AAA  BBB  CCC
0    4   10  100
```

2.22.2 Selection

DataFrames

The *indexing* docs.

Using both row labels and value conditionals

```
In [39]: df = pd.DataFrame({'AAA': [4, 5, 6, 7],
   ....:                    'BBB': [10, 20, 30, 40],
   ....:                    'CCC': [100, 50, -30, -50]})
   ....:

In [40]: df
Out[40]:
   AAA  BBB  CCC
0    4   10  100
1    5   20   50
```

(continues on next page)

(continued from previous page)

```
2    6   30  -30
3    7   40  -50

In [41]: df[(df.AAA <= 6) & (df.index.isin([0, 2, 4]))]
Out[41]:
   AAA  BBB  CCC
0    4   10  100
2    6   30  -30
```

Use loc for label-oriented slicing and iloc positional slicing

```
In [42]: df = pd.DataFrame({'AAA': [4, 5, 6, 7],
   ....:                    'BBB': [10, 20, 30, 40],
   ....:                    'CCC': [100, 50, -30, -50]},
   ....:                   index=['foo', 'bar', 'boo', 'kar'])
   ....:
```

There are 2 explicit slicing methods, with a third general case

1. Positional-oriented (Python slicing style : exclusive of end)
2. Label-oriented (Non-Python slicing style : inclusive of end)
3. General (Either slicing style : depends on if the slice contains labels or positions)

```
In [43]: df.loc['bar':'kar']  # Label
Out[43]:
     AAA  BBB  CCC
bar    5   20   50
boo    6   30  -30
kar    7   40  -50

# Generic
In [44]: df.iloc[0:3]
Out[44]:
     AAA  BBB  CCC
foo    4   10  100
bar    5   20   50
boo    6   30  -30

In [45]: df.loc['bar':'kar']
Out[45]:
     AAA  BBB  CCC
bar    5   20   50
boo    6   30  -30
kar    7   40  -50
```

Ambiguity arises when an index consists of integers with a non-zero start or non-unit increment.

```
In [46]: data = {'AAA': [4, 5, 6, 7],
   ....:         'BBB': [10, 20, 30, 40],
   ....:         'CCC': [100, 50, -30, -50]}
   ....:

In [47]: df2 = pd.DataFrame(data=data, index=[1, 2, 3, 4])  # Note index starts at 1.

In [48]: df2.iloc[1:3]  # Position-oriented
Out[48]:
```

(continues on next page)

(continued from previous page)

```
   AAA  BBB  CCC
2    5   20   50
3    6   30  -30

In [49]: df2.loc[1:3]  # Label-oriented
Out[49]:
   AAA  BBB  CCC
1    4   10  100
2    5   20   50
3    6   30  -30
```

Using inverse operator (~) to take the complement of a mask

```
In [50]: df = pd.DataFrame({'AAA': [4, 5, 6, 7],
   ....:                    'BBB': [10, 20, 30, 40],
   ....:                    'CCC': [100, 50, -30, -50]})
   ....:

In [51]: df
Out[51]:
   AAA  BBB  CCC
0    4   10  100
1    5   20   50
2    6   30  -30
3    7   40  -50

In [52]: df[~((df.AAA <= 6) & (df.index.isin([0, 2, 4])))]
Out[52]:
   AAA  BBB  CCC
1    5   20   50
3    7   40  -50
```

New columns

Efficiently and dynamically creating new columns using applymap

```
In [53]: df = pd.DataFrame({'AAA': [1, 2, 1, 3],
   ....:                    'BBB': [1, 1, 2, 2],
   ....:                    'CCC': [2, 1, 3, 1]})
   ....:

In [54]: df
Out[54]:
   AAA  BBB  CCC
0    1    1    2
1    2    1    1
2    1    2    3
3    3    2    1

In [55]: source_cols = df.columns   # Or some subset would work too

In [56]: new_cols = [str(x) + "_cat" for x in source_cols]

In [57]: categories = {1: 'Alpha', 2: 'Beta', 3: 'Charlie'}
```

(continues on next page)

(continued from previous page)

```
In [58]: df[new_cols] = df[source_cols].applymap(categories.get)

In [59]: df
Out[59]:
   AAA  BBB  CCC  AAA_cat BBB_cat  CCC_cat
0    1    1    2    Alpha   Alpha     Beta
1    2    1    1     Beta   Alpha    Alpha
2    1    2    3    Alpha    Beta  Charlie
3    3    2    1  Charlie    Beta    Alpha
```

Keep other columns when using min() with groupby

```
In [60]: df = pd.DataFrame({'AAA': [1, 1, 1, 2, 2, 2, 3, 3],
   ....:                    'BBB': [2, 1, 3, 4, 5, 1, 2, 3]})
   ....:

In [61]: df
Out[61]:
   AAA  BBB
0    1    2
1    1    1
2    1    3
3    2    4
4    2    5
5    2    1
6    3    2
7    3    3
```

Method 1 : idxmin() to get the index of the minimums

```
In [62]: df.loc[df.groupby("AAA")["BBB"].idxmin()]
Out[62]:
   AAA  BBB
1    1    1
5    2    1
6    3    2
```

Method 2 : sort then take first of each

```
In [63]: df.sort_values(by="BBB").groupby("AAA", as_index=False).first()
Out[63]:
   AAA  BBB
0    1    1
1    2    1
2    3    2
```

Notice the same results, with the exception of the index.

2.22.3 MultiIndexing

The *multiindexing* docs.

Creating a MultiIndex from a labeled frame

```
In [64]: df = pd.DataFrame({'row': [0, 1, 2],
   ....:                    'One_X': [1.1, 1.1, 1.1],
   ....:                    'One_Y': [1.2, 1.2, 1.2],
   ....:                    'Two_X': [1.11, 1.11, 1.11],
   ....:                    'Two_Y': [1.22, 1.22, 1.22]})
   ....:

In [65]: df
Out[65]:
   row  One_X  One_Y  Two_X  Two_Y
0    0    1.1    1.2   1.11   1.22
1    1    1.1    1.2   1.11   1.22
2    2    1.1    1.2   1.11   1.22

# As Labelled Index
In [66]: df = df.set_index('row')

In [67]: df
Out[67]:
     One_X  One_Y  Two_X  Two_Y
row
0      1.1    1.2   1.11   1.22
1      1.1    1.2   1.11   1.22
2      1.1    1.2   1.11   1.22

# With Hierarchical Columns
In [68]: df.columns = pd.MultiIndex.from_tuples([tuple(c.split('_'))
   ....:                                         for c in df.columns])
   ....:

In [69]: df
Out[69]:
     One        Two
       X    Y     X     Y
row
0    1.1  1.2  1.11  1.22
1    1.1  1.2  1.11  1.22
2    1.1  1.2  1.11  1.22

# Now stack & Reset
In [70]: df = df.stack(0).reset_index(1)

In [71]: df
Out[71]:
    level_1     X     Y
row
0       One  1.10  1.20
0       Two  1.11  1.22
1       One  1.10  1.20
1       Two  1.11  1.22
2       One  1.10  1.20
2       Two  1.11  1.22
```

(continues on next page)

(continued from previous page)

```
# And fix the labels (Notice the label 'level_1' got added automatically)
In [72]: df.columns = ['Sample', 'All_X', 'All_Y']

In [73]: df
Out[73]:
    Sample  All_X  All_Y
row
0      One   1.10   1.20
0      Two   1.11   1.22
1      One   1.10   1.20
1      Two   1.11   1.22
2      One   1.10   1.20
2      Two   1.11   1.22
```

Arithmetic

Performing arithmetic with a MultiIndex that needs broadcasting

```
In [74]: cols = pd.MultiIndex.from_tuples([(x, y) for x in ['A', 'B', 'C']
   ....:                                   for y in ['O', 'I']])
   ....:

In [75]: df = pd.DataFrame(np.random.randn(2, 6), index=['n', 'm'], columns=cols)

In [76]: df
Out[76]:
          A                   B                   C
          O         I         O         I         O         I
n  0.469112 -0.282863 -1.509059 -1.135632  1.212112 -0.173215
m  0.119209 -1.044236 -0.861849 -2.104569 -0.494929  1.071804

In [77]: df = df.div(df['C'], level=1)

In [78]: df
Out[78]:
          A                   B              C
          O         I         O         I    O    I
n  0.387021  1.633022 -1.244983  6.556214  1.0  1.0
m -0.240860 -0.974279  1.741358 -1.963577  1.0  1.0
```

Slicing

Slicing a MultiIndex with xs

```
In [79]: coords = [('AA', 'one'), ('AA', 'six'), ('BB', 'one'), ('BB', 'two'),
   ....:           ('BB', 'six')]
   ....:

In [80]: index = pd.MultiIndex.from_tuples(coords)

In [81]: df = pd.DataFrame([11, 22, 33, 44, 55], index, ['MyData'])

In [82]: df
```

(continues on next page)

(continued from previous page)

```
Out[82]:
        MyData
AA one      11
   six      22
BB one      33
   two      44
   six      55
```

To take the cross section of the 1st level and 1st axis the index:

```
# Note : level and axis are optional, and default to zero
In [83]: df.xs('BB', level=0, axis=0)
Out[83]:
     MyData
one      33
two      44
six      55
```

. . . and now the 2nd level of the 1st axis.

```
In [84]: df.xs('six', level=1, axis=0)
Out[84]:
    MyData
AA      22
BB      55
```

Slicing a MultiIndex with xs, method #2

```
In [85]: import itertools

In [86]: index = list(itertools.product(['Ada', 'Quinn', 'Violet'],
   ....:                                 ['Comp', 'Math', 'Sci']))
   ....:

In [87]: headr = list(itertools.product(['Exams', 'Labs'], ['I', 'II']))

In [88]: indx = pd.MultiIndex.from_tuples(index, names=['Student', 'Course'])

In [89]: cols = pd.MultiIndex.from_tuples(headr)   # Notice these are un-named

In [90]: data = [[70 + x + y + (x * y) % 3 for x in range(4)] for y in range(9)]

In [91]: df = pd.DataFrame(data, indx, cols)

In [92]: df
Out[92]:
               Exams     Labs
                   I  II    I  II
Student Course
Ada     Comp      70  71   72  73
        Math      71  73   75  74
        Sci       72  75   75  75
Quinn   Comp      73  74   75  76
        Math      74  76   78  77
        Sci       75  78   78  78
Violet  Comp      76  77   78  79
        Math      77  79   81  80
```

(continues on next page)

(continued from previous page)

```
        Sci        78  81   81  81

In [93]: All = slice(None)

In [94]: df.loc['Violet']
Out[94]:
       Exams     Labs
           I  II    I  II
Course
Comp      76  77   78  79
Math      77  79   81  80
Sci       78  81   81  81

In [95]: df.loc[(All, 'Math'), All]
Out[95]:
               Exams     Labs
                   I  II    I  II
Student Course
Ada     Math      71  73   75  74
Quinn   Math      74  76   78  77
Violet  Math      77  79   81  80

In [96]: df.loc[(slice('Ada', 'Quinn'), 'Math'), All]
Out[96]:
               Exams     Labs
                   I  II    I  II
Student Course
Ada     Math      71  73   75  74
Quinn   Math      74  76   78  77

In [97]: df.loc[(All, 'Math'), ('Exams')]
Out[97]:
                 I  II
Student Course
Ada     Math    71  73
Quinn   Math    74  76
Violet  Math    77  79

In [98]: df.loc[(All, 'Math'), (All, 'II')]
Out[98]:
               Exams Labs
                  II   II
Student Course
Ada     Math      73   74
Quinn   Math      76   77
Violet  Math      79   80
```

Setting portions of a MultiIndex with xs

Sorting

Sort by specific column or an ordered list of columns, with a MultiIndex

```
In [99]: df.sort_values(by=('Labs', 'II'), ascending=False)
Out[99]:
                Exams     Labs
                    I  II    I  II
Student Course
Violet  Sci        78  81   81  81
        Math       77  79   81  80
        Comp       76  77   78  79
Quinn   Sci        75  78   78  78
        Math       74  76   78  77
        Comp       73  74   75  76
Ada     Sci        72  75   75  75
        Math       71  73   75  74
        Comp       70  71   72  73
```

Partial selection, the need for sortedness;

Levels

Prepending a level to a multiindex

Flatten Hierarchical columns

2.22.4 Missing data

The *missing data* docs.

Fill forward a reversed timeseries

```
In [100]: df = pd.DataFrame(np.random.randn(6, 1),
   .....:                   index=pd.date_range('2013-08-01', periods=6, freq='B'),
   .....:                   columns=list('A'))
   .....:

In [101]: df.loc[df.index[3], 'A'] = np.nan

In [102]: df
Out[102]:
                   A
2013-08-01  0.721555
2013-08-02 -0.706771
2013-08-05 -1.039575
2013-08-06       NaN
2013-08-07 -0.424972
2013-08-08  0.567020

In [103]: df.reindex(df.index[::-1]).ffill()
Out[103]:
                   A
2013-08-08  0.567020
2013-08-07 -0.424972
2013-08-06 -0.424972
```

(continues on next page)

(continued from previous page)

```
2013-08-05 -1.039575
2013-08-02 -0.706771
2013-08-01  0.721555
```

cumsum reset at NaN values

Replace

Using replace with backrefs

2.22.5 Grouping

The *grouping* docs.

Basic grouping with apply

Unlike agg, apply's callable is passed a sub-DataFrame which gives you access to all the columns

```
In [104]: df = pd.DataFrame({'animal': 'cat dog cat fish dog cat cat'.split(),
   .....:                    'size': list('SSMMMLL'),
   .....:                    'weight': [8, 10, 11, 1, 20, 12, 12],
   .....:                    'adult': [False] * 5 + [True] * 2})
   .....:

In [105]: df
Out[105]:
  animal size  weight  adult
0    cat    S       8  False
1    dog    S      10  False
2    cat    M      11  False
3   fish    M       1  False
4    dog    M      20  False
5    cat    L      12   True
6    cat    L      12   True

# List the size of the animals with the highest weight.
In [106]: df.groupby('animal').apply(lambda subf: subf['size'][subf['weight'].
↪idxmax()])
Out[106]:
animal
cat     L
dog     M
fish    M
dtype: object
```

Using get_group

```
In [107]: gb = df.groupby(['animal'])

In [108]: gb.get_group('cat')
Out[108]:
  animal size  weight  adult
0    cat    S       8  False
2    cat    M      11  False
```

(continues on next page)

(continued from previous page)

```
5     cat     L       12    True
6     cat     L       12    True
```

Apply to different items in a group

```
In [109]: def GrowUp(x):
   .....:     avg_weight = sum(x[x['size'] == 'S'].weight * 1.5)
   .....:     avg_weight += sum(x[x['size'] == 'M'].weight * 1.25)
   .....:     avg_weight += sum(x[x['size'] == 'L'].weight)
   .....:     avg_weight /= len(x)
   .....:     return pd.Series(['L', avg_weight, True],
   .....:                      index=['size', 'weight', 'adult'])
   .....:

In [110]: expected_df = gb.apply(GrowUp)

In [111]: expected_df
Out[111]:
       size   weight  adult
animal
cat       L  12.4375   True
dog       L  20.0000   True
fish      L   1.2500   True
```

Expanding apply

```
In [112]: S = pd.Series([i / 100.0 for i in range(1, 11)])

In [113]: def cum_ret(x, y):
   .....:     return x * (1 + y)
   .....:

In [114]: def red(x):
   .....:     return functools.reduce(cum_ret, x, 1.0)
   .....:

In [115]: S.expanding().apply(red, raw=True)
Out[115]:
0    1.010000
1    1.030200
2    1.061106
3    1.103550
4    1.158728
5    1.228251
6    1.314229
7    1.419367
8    1.547110
9    1.701821
dtype: float64
```

Replacing some values with mean of the rest of a group

```
In [116]: df = pd.DataFrame({'A': [1, 1, 2, 2], 'B': [1, -1, 1, 2]})

In [117]: gb = df.groupby('A')

In [118]: def replace(g):
```

(continues on next page)

(continued from previous page)

```
   .....:     mask = g < 0
   .....:     return g.where(mask, g[~mask].mean())
   .....:

In [119]: gb.transform(replace)
Out[119]:
     B
0  1.0
1 -1.0
2  1.5
3  1.5
```

Sort groups by aggregated data

```
In [120]: df = pd.DataFrame({'code': ['foo', 'bar', 'baz'] * 2,
   .....:                    'data': [0.16, -0.21, 0.33, 0.45, -0.59, 0.62],
   .....:                    'flag': [False, True] * 3})
   .....:

In [121]: code_groups = df.groupby('code')

In [122]: agg_n_sort_order = code_groups[['data']].transform(sum).sort_values(by='data
↪')

In [123]: sorted_df = df.loc[agg_n_sort_order.index]

In [124]: sorted_df
Out[124]:
  code  data   flag
1  bar -0.21   True
4  bar -0.59  False
0  foo  0.16  False
3  foo  0.45   True
2  baz  0.33  False
5  baz  0.62   True
```

Create multiple aggregated columns

```
In [125]: rng = pd.date_range(start="2014-10-07", periods=10, freq='2min')

In [126]: ts = pd.Series(data=list(range(10)), index=rng)

In [127]: def MyCust(x):
   .....:     if len(x) > 2:
   .....:         return x[1] * 1.234
   .....:     return pd.NaT
   .....:

In [128]: mhc = {'Mean': np.mean, 'Max': np.max, 'Custom': MyCust}

In [129]: ts.resample("5min").apply(mhc)
Out[129]:
Mean    2014-10-07 00:00:00        1
        2014-10-07 00:05:00      3.5
        2014-10-07 00:10:00        6
        2014-10-07 00:15:00      8.5
Max     2014-10-07 00:00:00        2
```

(continues on next page)

(continued from previous page)

```
        2014-10-07 00:05:00        4
        2014-10-07 00:10:00        7
        2014-10-07 00:15:00        9
Custom  2014-10-07 00:00:00    1.234
        2014-10-07 00:05:00      NaT
        2014-10-07 00:10:00    7.404
        2014-10-07 00:15:00      NaT
dtype: object

In [130]: ts
Out[130]:
2014-10-07 00:00:00    0
2014-10-07 00:02:00    1
2014-10-07 00:04:00    2
2014-10-07 00:06:00    3
2014-10-07 00:08:00    4
2014-10-07 00:10:00    5
2014-10-07 00:12:00    6
2014-10-07 00:14:00    7
2014-10-07 00:16:00    8
2014-10-07 00:18:00    9
Freq: 2T, dtype: int64
```

Create a value counts column and reassign back to the DataFrame

```
In [131]: df = pd.DataFrame({'Color': 'Red Red Red Blue'.split(),
   .....:                    'Value': [100, 150, 50, 50]})
   .....:

In [132]: df
Out[132]:
  Color  Value
0   Red    100
1   Red    150
2   Red     50
3  Blue     50

In [133]: df['Counts'] = df.groupby(['Color']).transform(len)

In [134]: df
Out[134]:
  Color  Value  Counts
0   Red    100       3
1   Red    150       3
2   Red     50       3
3  Blue     50       1
```

Shift groups of the values in a column based on the index

```
In [135]: df = pd.DataFrame({'line_race': [10, 10, 8, 10, 10, 8],
   .....:                    'beyer': [99, 102, 103, 103, 88, 100]},
   .....:                   index=['Last Gunfighter', 'Last Gunfighter',
   .....:                          'Last Gunfighter', 'Paynter', 'Paynter',
   .....:                          'Paynter'])
   .....:

In [136]: df
```

(continues on next page)

(continued from previous page)

```
Out[136]:
                  line_race  beyer
Last Gunfighter          10     99
Last Gunfighter          10    102
Last Gunfighter           8    103
Paynter                  10    103
Paynter                  10     88
Paynter                   8    100

In [137]: df['beyer_shifted'] = df.groupby(level=0)['beyer'].shift(1)

In [138]: df
Out[138]:
                  line_race  beyer  beyer_shifted
Last Gunfighter          10     99            NaN
Last Gunfighter          10    102           99.0
Last Gunfighter           8    103          102.0
Paynter                  10    103            NaN
Paynter                  10     88          103.0
Paynter                   8    100           88.0
```

Select row with maximum value from each group

```
In [139]: df = pd.DataFrame({'host': ['other', 'other', 'that', 'this', 'this'],
   .....:                    'service': ['mail', 'web', 'mail', 'mail', 'web'],
   .....:                    'no': [1, 2, 1, 2, 1]}).set_index(['host', 'service'])
   .....:

In [140]: mask = df.groupby(level=0).agg('idxmax')

In [141]: df_count = df.loc[mask['no']].reset_index()

In [142]: df_count
Out[142]:
    host service  no
0  other     web   2
1   that    mail   1
2   this    mail   2
```

Grouping like Python's itertools.groupby

```
In [143]: df = pd.DataFrame([0, 1, 0, 1, 1, 1, 0, 1, 1], columns=['A'])

In [144]: df['A'].groupby((df['A'] != df['A'].shift()).cumsum()).groups
Out[144]:
{1: Int64Index([0], dtype='int64'),
 2: Int64Index([1], dtype='int64'),
 3: Int64Index([2], dtype='int64'),
 4: Int64Index([3, 4, 5], dtype='int64'),
 5: Int64Index([6], dtype='int64'),
 6: Int64Index([7, 8], dtype='int64')}

In [145]: df['A'].groupby((df['A'] != df['A'].shift()).cumsum()).cumsum()
Out[145]:
0    0
1    1
2    0
```

(continues on next page)

(continued from previous page)

```
3    1
4    2
5    3
6    0
7    1
8    2
Name: A, dtype: int64
```

Expanding data

Alignment and to-date

Rolling Computation window based on values instead of counts

Rolling Mean by Time Interval

Splitting

Splitting a frame

Create a list of dataframes, split using a delineation based on logic included in rows.

```
In [146]: df = pd.DataFrame(data={'Case': ['A', 'A', 'A', 'B', 'A', 'A', 'B', 'A',
   .....:                                  'A'],
   .....:                         'Data': np.random.randn(9)})
   .....:

In [147]: dfs = list(zip(*df.groupby((1 * (df['Case'] == 'B')).cumsum()
   .....:                 .rolling(window=3, min_periods=1).median())))[-1]
   .....:

In [148]: dfs[0]
Out[148]:
  Case      Data
0    A  0.276232
1    A -1.087401
2    A -0.673690
3    B  0.113648

In [149]: dfs[1]
Out[149]:
  Case      Data
4    A -1.478427
5    A  0.524988
6    B  0.404705

In [150]: dfs[2]
Out[150]:
  Case      Data
7    A  0.577046
8    A -1.715002
```

Pivot

The *Pivot* docs.

Partial sums and subtotals

```
In [151]: df = pd.DataFrame(data={'Province': ['ON', 'QC', 'BC', 'AL', 'AL', 'MN', 'ON
↪'],
   .....:                         'City': ['Toronto', 'Montreal', 'Vancouver',
   .....:                                  'Calgary', 'Edmonton', 'Winnipeg',
   .....:                                  'Windsor'],
   .....:                         'Sales': [13, 6, 16, 8, 4, 3, 1]})
   .....:

In [152]: table = pd.pivot_table(df, values=['Sales'], index=['Province'],
   .....:                        columns=['City'], aggfunc=np.sum, margins=True)
   .....:

In [153]: table.stack('City')
Out[153]:
                    Sales
Province City
AL       All         12.0
         Calgary      8.0
         Edmonton     4.0
BC       All         16.0
         Vancouver   16.0
...                   ...
All      Montreal     6.0
         Toronto     13.0
         Vancouver   16.0
         Windsor      1.0
         Winnipeg     3.0

[20 rows x 1 columns]
```

Frequency table like plyr in R

```
In [154]: grades = [48, 99, 75, 80, 42, 80, 72, 68, 36, 78]

In [155]: df = pd.DataFrame({'ID': ["x%d" % r for r in range(10)],
   .....:                    'Gender': ['F', 'M', 'F', 'M', 'F',
   .....:                               'M', 'F', 'M', 'M', 'M'],
   .....:                    'ExamYear': ['2007', '2007', '2007', '2008', '2008',
   .....:                                 '2008', '2008', '2009', '2009', '2009'],
   .....:                    'Class': ['algebra', 'stats', 'bio', 'algebra',
   .....:                              'algebra', 'stats', 'stats', 'algebra',
   .....:                              'bio', 'bio'],
   .....:                    'Participated': ['yes', 'yes', 'yes', 'yes', 'no',
   .....:                                     'yes', 'yes', 'yes', 'yes', 'yes'],
   .....:                    'Passed': ['yes' if x > 50 else 'no' for x in grades],
   .....:                    'Employed': [True, True, True, False,
   .....:                                 False, False, False, True, True, False],
   .....:                    'Grade': grades})
   .....:

In [156]: df.groupby('ExamYear').agg({'Participated': lambda x: x.value_counts()['yes
↪'],
```

(continues on next page)

(continued from previous page)

```
   .....:                                  'Passed': lambda x: sum(x == 'yes'),
   .....:                                  'Employed': lambda x: sum(x),
   .....:                                  'Grade': lambda x: sum(x) / len(x)})
   .....:
Out[156]:
          Participated  Passed  Employed      Grade
ExamYear
2007                 3       2         3  74.000000
2008                 3       3         0  68.500000
2009                 3       2         2  60.666667
```

Plot pandas DataFrame with year over year data

To create year and month cross tabulation:

```
In [157]: df = pd.DataFrame({'value': np.random.randn(36)},
   .....:                   index=pd.date_range('2011-01-01', freq='M', periods=36))
   .....:

In [158]: pd.pivot_table(df, index=df.index.month, columns=df.index.year,
   .....:                values='value', aggfunc='sum')
   .....:
Out[158]:
        2011      2012      2013
1  -1.039268 -0.968914  2.565646
2  -0.370647 -1.294524  1.431256
3  -1.157892  0.413738  1.340309
4  -1.344312  0.276662 -1.170299
5   0.844885 -0.472035 -0.226169
6   1.075770 -0.013960  0.410835
7  -0.109050 -0.362543  0.813850
8   1.643563 -0.006154  0.132003
9  -1.469388 -0.923061 -0.827317
10  0.357021  0.895717 -0.076467
11 -0.674600  0.805244 -1.187678
12 -1.776904 -1.206412  1.130127
```

Apply

Rolling apply to organize - Turning embedded lists into a MultiIndex frame

```
In [159]: df = pd.DataFrame(data={'A': [[2, 4, 8, 16], [100, 200], [10, 20, 30]],
   .....:                         'B': [['a', 'b', 'c'], ['jj', 'kk'], ['ccc']]},
   .....:                   index=['I', 'II', 'III'])
   .....:

In [160]: def SeriesFromSubList(aList):
   .....:     return pd.Series(aList)
   .....:

In [161]: df_orgz = pd.concat({ind: row.apply(SeriesFromSubList)
   .....:                      for ind, row in df.iterrows()})
   .....:

In [162]: df_orgz
```

(continues on next page)

(continued from previous page)

```
Out[162]:
          0    1    2     3
I   A     2    4    8  16.0
    B     a    b    c   NaN
II  A   100  200  NaN   NaN
    B    jj   kk  NaN   NaN
III A    10   20   30   NaN
    B   ccc  NaN  NaN   NaN
```

Rolling apply with a DataFrame returning a Series

Rolling Apply to multiple columns where function calculates a Series before a Scalar from the Series is returned

```
In [163]: df = pd.DataFrame(data=np.random.randn(2000, 2) / 10000,
   .....:                   index=pd.date_range('2001-01-01', periods=2000),
   .....:                   columns=['A', 'B'])
   .....:

In [164]: df
Out[164]:
                   A         B
2001-01-01 -0.000144 -0.000141
2001-01-02  0.000161  0.000102
2001-01-03  0.000057  0.000088
2001-01-04 -0.000221  0.000097
2001-01-05 -0.000201 -0.000041
...              ...       ...
2006-06-19  0.000040 -0.000235
2006-06-20 -0.000123 -0.000021
2006-06-21 -0.000113  0.000114
2006-06-22  0.000136  0.000109
2006-06-23  0.000027  0.000030

[2000 rows x 2 columns]

In [165]: def gm(df, const):
   .....:     v = ((((df['A'] + df['B']) + 1).cumprod()) - 1) * const
   .....:     return v.iloc[-1]
   .....:

In [166]: s = pd.Series({df.index[i]: gm(df.iloc[i:min(i + 51, len(df) - 1)], 5)
   .....:                for i in range(len(df) - 50)})
   .....:

In [167]: s
Out[167]:
2001-01-01    0.000930
2001-01-02    0.002615
2001-01-03    0.001281
2001-01-04    0.001117
2001-01-05    0.002772
                ...
2006-04-30    0.003296
2006-05-01    0.002629
2006-05-02    0.002081
2006-05-03    0.004247
2006-05-04    0.003928
```

(continues on next page)

(continued from previous page)

```
Length: 1950, dtype: float64
```

Rolling apply with a DataFrame returning a Scalar

Rolling Apply to multiple columns where function returns a Scalar (Volume Weighted Average Price)

```
In [168]: rng = pd.date_range(start='2014-01-01', periods=100)

In [169]: df = pd.DataFrame({'Open': np.random.randn(len(rng)),
   .....:                    'Close': np.random.randn(len(rng)),
   .....:                    'Volume': np.random.randint(100, 2000, len(rng))},
   .....:                   index=rng)
   .....:

In [170]: df
Out[170]:
                Open     Close  Volume
2014-01-01 -1.611353 -0.492885    1219
2014-01-02 -3.000951  0.445794    1054
2014-01-03 -0.138359 -0.076081    1381
2014-01-04  0.301568  1.198259    1253
2014-01-05  0.276381 -0.669831    1728
...              ...       ...     ...
2014-04-06 -0.040338  0.937843    1188
2014-04-07  0.359661 -0.285908    1864
2014-04-08  0.060978  1.714814     941
2014-04-09  1.759055 -0.455942    1065
2014-04-10  0.138185 -1.147008    1453

[100 rows x 3 columns]

In [171]: def vwap(bars):
   .....:     return ((bars.Close * bars.Volume).sum() / bars.Volume.sum())
   .....:

In [172]: window = 5

In [173]: s = pd.concat([(pd.Series(vwap(df.iloc[i:i + window]),
   .....:                 index=[df.index[i + window]]))
   .....:                for i in range(len(df) - window)])
   .....:

In [174]: s.round(2)
Out[174]:
2014-01-06    0.02
2014-01-07    0.11
2014-01-08    0.10
2014-01-09    0.07
2014-01-10   -0.29
              ...
2014-04-06   -0.63
2014-04-07   -0.02
2014-04-08   -0.03
2014-04-09    0.34
2014-04-10    0.29
Length: 95, dtype: float64
```

2.22.6 Timeseries

Between times

Using indexer between time

Constructing a datetime range that excludes weekends and includes only certain times

Vectorized Lookup

Aggregation and plotting time series

Turn a matrix with hours in columns and days in rows into a continuous row sequence in the form of a time series. How to rearrange a Python pandas DataFrame?

Dealing with duplicates when reindexing a timeseries to a specified frequency

Calculate the first day of the month for each entry in a DatetimeIndex

```
In [175]: dates = pd.date_range('2000-01-01', periods=5)

In [176]: dates.to_period(freq='M').to_timestamp()
Out[176]:
DatetimeIndex(['2000-01-01', '2000-01-01', '2000-01-01', '2000-01-01',
               '2000-01-01'],
              dtype='datetime64[ns]', freq=None)
```

Resampling

The *Resample* docs.

Using Grouper instead of TimeGrouper for time grouping of values

Time grouping with some missing values

Valid frequency arguments to Grouper

Grouping using a MultiIndex

Using TimeGrouper and another grouping to create subgroups, then apply a custom function

Resampling with custom periods

Resample intraday frame without adding new days

Resample minute data

Resample with groupby

2.22.7 Merge

The *Concat* docs. The *Join* docs.

Append two dataframes with overlapping index (emulate R rbind)

```
In [177]: rng = pd.date_range('2000-01-01', periods=6)

In [178]: df1 = pd.DataFrame(np.random.randn(6, 3), index=rng, columns=['A', 'B', 'C
↪'])

In [179]: df2 = df1.copy()
```