INTRODUCING PANDAS OBJECTS

Pandas objects can be thought of as enhanced versions of NumPy structured arrays in which the rows and columns are identified with labels rather than simple integer indices.

Three fundamental Pandas data structures: the Series, DataFrame, and Index.

```
import pandas as pd
```

The Pandas Series Object

A Pandas Series is a one-dimensional array of indexed data. It can be created from a

list or array as follows:

```
data = pd.Series([0.25, 0.5, 0.75, 1.0])
data
```

```
0      0.25
1      0.50
2      0.75
3      1.00
dtype: float64
```

data.values

array([0.25, 0.5 , 0.75, 1.])

The index is an array-like object of type pd.Index

RangeIndex(start=0, stop=4, step=1)

Like with a NumPy array, data can be accessed by the associated index via the familiar

Python square-bracket notation:

data[1]

0.5

data[1:3]

1 0.50

2 0.75

dtype: float64

Series as generalized NumPy array

```
data = pd.Series([0.25, 0.5, 0.75, 1.0],index=['a', 'b', 'c', 'd'])
Data
 a    0.25
b    0.50
c    0.75
d    1.00
dtype: float64
```

And the item access works as expected:

```
data['b']
 0.5
```

We can even use noncontiguous or nonsequential indices:

```
data = pd.Series([0.25, 0.5, 0.75, 1.0],index=[2, 5, 3, 7])
data
2    0.25
5    0.50
3    0.75
7    1.00
dtype: float64
```

Series as specialized dictionary

A dictionary is a structure that maps arbitrary keys to a set of arbitrary values, and a Series is a structure that maps typed keys to a set of typed values.

```
population_dict = {'California': 38332521,
'Texas': 26448193,
'New York': 19651127,
'Florida': 19552860,
'Illinois': 12882135}
population = pd.Series(population_dict)
Population
```

```
California      38332521
Florida         19552860
Illinois        12882135
New York        19651127
Texas           26448193
dtype: int64
```

The Pandas DataFrame Object

the DataFrame can be thought of either as a generalization

of a NumPy array, or as a specialization of a Python dictionary

DataFrame as a generalized NumPy array

A DataFrame is an analog of a two-dimensional array with both flexible row indices and flexible column names.

```
area_dict = {'California': 423967, 'Texas': 695662, 'New York': 141297,
'Florida': 170312, 'Illinois': 149995}
area = pd.Series(area_dict)
area
California      423967
Florida         170312
Illinois        149995
New York        141297
Texas           695662
dtype: int6
```

```
states = pd.DataFrame({'population': population,
'area': area})
states
```

	area	population
California	423967	38332521
Florida	170312	19552860
Illinois	149995	12882135
New York	141297	19651127
Texas	695662	26448193

```python
data = [{'a': i, 'b': 2 * i}
for i in range(3)]
pd.DataFrame(data)
```

```
   a  b
0  0  0
1  1  2
2  2  4
```

Even if some keys in the dictionary are missing, Pandas will fill them in with NaN (i.e.,

“not a number”) values:

```python
pd.DataFrame([{'a': 1, 'b': 2}, {'b': 3, 'c': 4}])
```

```
     a    b   c
0  1.0    2  NaN
1  NaN    3  4.0
```

The Pandas Index Object

Index object is an interesting structure in itself, and it can be thought of either as an *immutable array* or as an *ordered set.*

```
ind = pd.Index([2, 3, 5, 7, 11])
```

```
Ind
```

Int64Index([2, 3, 5, 7, 11], dtype='int64’)

Index as immutable array

The Index object in many ways operates like an array. For example, we can use standard

Python indexing notation to retrieve values or slices:

```
ind[1]
```

3

```
ind[::2]
```

Int64Index([2, 5, 11], dtype='int64')

Index as ordered set

Pandas objects are designed to facilitate operations such as joins across datasets,

which depend on many aspects of set arithmetic.

The Index object follows many of the conventions used by Python’s built-in set data structure, so that unions, intersections,

differences, and other combinations can be computed in a familiar way:

```
indA = pd.Index([1, 3, 5, 7, 9])
indB = pd.Index([2, 3, 5, 7, 11])
indA & indB # intersection
Int64Index([3, 5, 7], dtype='int64')
indA | indB # union
Int64Index([1, 2, 3, 5, 7, 9, 11], dtype='int64')
indA ^ indB # symmetric difference
Int64Index([1, 2, 9, 11], dtype='int64')
```

DATA INDEXING AND SELECTION

Data Selection in Series

Series as dictionary

Like a dictionary, the Series object provides a mapping from a collection of keys to a collection of values:

```
import pandas as pd
data = pd.Series([0.25, 0.5, 0.75, 1.0],
index=['a', 'b', 'c', 'd'])
data
```

```
Data
a    0.25
b    0.50
c    0.75
d    1.00
dtype: float64
```

We can also use dictionary-like Python expressions and methods to examine the keys/indices and values:

```
'a' in data
```

True

```
data.keys()
```

Index(['a', 'b', 'c', 'd'], dtype='object')

```
list(data.items())
```

[('a', 0.25), ('b', 0.5), ('c', 0.75), ('d', 1.0)]

Series objects can even be modified with a dictionary-like syntax.

Series as one-dimensional array

A Series builds on this dictionary-like interface and provides array-style item selection

via the same basic mechanisms as NumPy arrays—that is, *slices*, *masking*, and

fancy indexing. Examples of these are as follows:

```
# slicing by explicit index
data['a':'c']
a    0.25
b    0.50
c    0.75
dtype: float64
```

```
# slicing by implicit integer index
data[0:2]
a    0.25
b    0.50
dtype: float64
# masking
data[(data > 0.3) & (data < 0.8)]
b    0.50
c    0.75
dtype: float64
# fancy indexing
data[['a', 'e']]
a    0.25
e    1.25
```

Indexers: loc, iloc, and ix

These slicing and indexing conventions can be a source of confusion.

```
data = pd.Series(['a', 'b', 'c'], index=[1, 3, 5])
data
 1    a
3    b
5    c
dtype: object
 # explicit index when indexing
data[1]
 'a'
 # implicit index when slicing
data[1:3]
 3     b
5     c
dtype: object
```

Because of this potential confusion in the case of integer indexes, Pandas provides some special *indexer* attributes that explicitly expose certain indexing schemes.

the loc attribute allows indexing and slicing that always references the explicit

index: data.loc[1]

'a'

data.loc[1:3]

1 a

3 b

dtype: object

The iloc attribute allows indexing and slicing that always references the implicit

Python-style index:

data.iloc[1]

'b'

data.iloc[1:3]

3 b

5 c

dtype: object

A third indexing attribute, ix, is a hybrid of the two, and for Series objects is equivalent

to standard []-based indexing.

Data Selection in DataFrame

DataFrame as a dictionary

```
area = pd.Series({'California': 423967, 'Texas': 695662,
'New York': 141297, 'Florida': 170312,
'Illinois': 149995})
pop = pd.Series({'California': 38332521, 'Texas': 26448193,
'New York': 19651127, 'Florida': 19552860,
'Illinois': 12882135})
data = pd.DataFrame({'area':area, 'pop':pop})
Data
```

	area	pop
California	423967	38332521
Florida	170312	19552860
Illinois	149995	12882135
New York	141297	19651127
Texas	695662	26448193

```
data.area
California        423967
Florida           170312
Illinois          149995
New York          141297
Texas             695662
Name: area, dtype: int64

data.area is data['area']
 True

data.pop is data['pop']
False
```

DataFrame as two-dimensional array

the DataFrame as an enhanced twodimensional array. We can examine the raw underlying data array using the values

attribute:

```
data.values
```

```
array([[ 4.23967000e+05, 3.83325210e+07, 9.04139261e+01],
       [ 1.70312000e+05, 1.95528600e+07, 1.14806121e+02],
       [ 1.49995000e+05, 1.28821350e+07, 8.58837628e+01],
       [ 1.41297000e+05, 1.96511270e+07, 1.39076746e+02],
       [ 6.95662000e+05, 2.64481930e+07, 3.80187404e+01]])
```

Transpose the full DataFrame to swap rows and columns:

```
data.T
```

	California	Florida	Illinois	New York	Texas
area	4.239670e+05	1.703120e+05	1.499950e+05	1.412970e+05	6.956620e+05
Pop	3.833252e+07	1.955286e+07	1.288214e+07	1.965113e+07	2.644819e+07
density	9.041393e+01	1.148061e+02	8.588376e+01	1.390767e+02	3.801874e+01

When it comes to indexing of DataFrame objects, however, it is clear that the dictionary-style indexing of columns precludes our ability to simply treat it as a NumPy array.

passing a single index to an array accesses a row:

```
data.values[0]
```

```
array([ 4.23967000e+05, 3.83325210e+07, 9.04139261e+01])
```

```
data.iloc[:3, :2]
```

	area	pop
California	423967	38332521
Florida	170312	19552860
Illinois	149995	12882135

```
data.loc[:'Illinois', :'pop’]
```

	area	pop
California	423967	38332521
Florida	170312	19552860
Illinois	149995	12882135

The ix indexer allows a hybrid of these two approaches:

```
data.ix[:3, :'pop’]
```

	area	pop
California	423967	38332521
Florida	170312	19552860
Illinois	149995	12882135

Keep in mind that for integer indices, the ix indexer is subject to the same potential sources of confusion .Any of the familiar NumPy-style data access patterns can be used within these indexers.

loc indexer we can combine masking and fancy indexing as

```
data.loc[data.density > 100, ['pop', 'density’]]
```

	pop	density
Florida	19552860	114.806121
New York	19651127	139.076746

Operating on Data in Pandas

These ufuncs will *preserve index and column labels* in the output, and for binary operations such as addition and multiplication,

Pandas will automatically *align indices* when passing the objects to the ufunc.

This means that keeping the context of data and combining data from different sources—both potentiallyerror-prone tasks with raw NumPy arrays—become essentially foolproof ones with Pandas.

We will additionally see that there are well-defined operations between one-dimensional Series structures and two-dimensional DataFrame structures.

Ufuncs: Index Preservation

```
import pandas as pd
import numpy as np
rng = np.random.RandomState(42)
ser = pd.Series(rng.randint(0, 10, 4))
Ser
 0  6
1   3
2   7
3   4
dtype: int64
df = pd.DataFrame(rng.randint(0, 10, (3, 4)),columns=['A', 'B', 'C', 'D’])
     A   B   C   D
0    6   9   2   6
1    7   4   3   7
2    7   2   5   4
```

UFuncs: Index Alignment

For binary operations on two Series or DataFrame objects, Pandas will align indices in the process of performing the operation.

Index alignment in Series

```
area = pd.Series({'Alaska': 1723337, 'Texas': 695662,'California': 423967}, name='area')
population = pd.Series({'California': 38332521, 'Texas': 26448193, 'New York': 19651127},
name='population’)
```

```
population / area
Alaska        NaN
California    90.413926
New York      NaN
Texas         38.018740
dtype: float64
```

Index alignment in Data frame

A similar typIndex alignment in DataFramee of alignment takes place for *both* columns and indices when you are

performing operations on DataFrames:

```
A = pd.DataFrame(rng.randint(0, 20, (2, 2)),
columns=list('AB'))
A
```

	A	B
0	1	11
1	5	1

```python
B = pd.DataFrame(rng.randint(0, 10, (3, 3)),
columns=list('BAC'))
B
```

	B	A	C
0	4	0	9
1	5	8	0
2	9	2	6

```python
A + B
```

	A	B	C
0	1.0	15.0	NaN
1	13.0	6.0	NaN
2	NaN	NaN	NaN

Index alignment in DataFrame

A similar type of alignment takes place for *both* columns and indices when you are performing operations on DataFrames:

```
A = pd.DataFrame(rng.randint(0, 20, (2, 2)),
columns=list('AB'))
A
```

	A	B
0	1	11
1	5	1

```
B = pd.DataFrame(rng.randint(0, 10, (3, 3)),
columns=list('BAC'))
B
```

	B	A	C
0	4	0	9
1	5	8	0
2	9	2	6

A + B

	A	B	C
0	1.0	15.0	NaN
1	13.0	6.0	NaN
2	NaN	NaN	NaN

Mapping between Python operators and Pandas methods

+	add()
-	sub(), subtract()
*	mul(), multiply()
/	truediv(), div(), divide()
//	floordiv()
%	mod()
**	pow()

Ufuncs: Operations Between DataFrame and Series

The index and column alignment is similarly maintained.

Operations between a DataFrame and a Series are similar to operations between a two-dimensional and one-dimensional NumPy array

```
A = rng.randint(10, size=(3, 4))
A
```

```
array([[3, 8, 2, 4],
       [2, 6, 4, 8],
       [6, 1, 3, 8]])
```

```
A - A[0]
```

```
array([[ 0, 0, 0, 0],
       [-1, -2, 2, 4],
       [ 3, -7, 1, 4]])
```

In Pandas, the convention similarly operates row-wise by default:

```
df = pd.DataFrame(A, columns=list('QRST'))
df - df.iloc[0]
```

	Q	R	S	T
0	0	0	0	0
1	-1	-2	2	4
2	3	-7	1	4

If you would instead like to operate column-wise, you can use the object methods mentioned earlier, while specifying the axis keyword:

```
df.subtract(df['R'], axis=0)
```

	Q	R	S	T
0	-5	0	-6	-4
1	-4	0	-2	2
2	5	0	2	7

HANDLING MISSING DATA

some built-in Pandas tools for handling missing data *null*, *NaN*, or *NA* values.

Trade-Offs in Missing Data Conventions

two strategies: using a *mask* that globally indicates missing values, or choosing a *sentinel value* that indicates

a missing entry.

In the **masking approach**, the mask might be an entirely separate Boolean array, or it may involve appropriation of one bit in the data representation to locally indicate the null status of a value.

In the **sentinel approach**, the sentinel value could be some data-specific convention, such as indicating a missing integer value with –9999 or some rare bit pattern, or it could be a more global convention, such as indicating a missing floating-point value with NaN (Not a Number), a special value which is part of the IEEE floating-point specification.

MISSING DATA IN PANDAS

Pandas chose to use sentinels for missing data, and chose to use two already-existing Python null values: the special floatingpoint NaN value, and the Python None object.

None: Pythonic missing data

```
import numpy as np
import pandas as pd
vals1 = np.array([1, None, 3, 4])
vals1
```

Output : array([1, None, 3, 4], dtype=object)

```
vals1.sum()
```

Output : TypeError

NaN: Missing numerical data

```
vals2 = np.array([1, np.nan, 3, 4])
vals2.dtype
```

Output: dtype('float64’)

the result of arithmetic with NaN will be another NaN:

```
1 + np.nan
```

Output: nan

```
0 * np.nan
```

Output: nan

```
vals2.sum(), vals2.min(), vals2.max()
```

Output: (nan, nan, nan)

```
np.nansum(vals2), np.nanmin(vals2), np.nanmax(vals2)
```

Output: (8.0, 1.0, 4.0)

NaN and None in Pandas

```
pd.Series([1, np.nan, 2, None])
```

Output: 0 1.0

1 NaN

2 2.0

3 NaN

dtype: float64

```
x = pd.Series(range(2), dtype=int)
x
```

Output: 0 0

1 1

dtype: int64

```
x[0] = None
x
```

Output: 0 NaN

1 1.0

dtype: float64

Pandas handling of NAs by type

floating	No change	np.nan
object	No change	None or np.nan
integer	Cast to float64	np.nan
boolean	Cast to object	None or np.nan

string data is always stored

OPERATING ON NULL VALUES

isnull()

Generate a Boolean mask indicating missing values

notnull()

Opposite of isnull()

dropna()

Return a filtered version of the data

fillna()

Return a copy of the data with missing values filled or imputed

DETECTING NULL VALUES

```
data = pd.Series([1, np.nan, 'hello', None])
data.isnull()
```

Output

```
0       False
1       True
2       False
3       True
dtype: bool
```

```
data[data.notnull()]
```

Output

```
0       1
2       hello
dtype: object
```

DROPPING NULL VALUES

```
data.dropna()
0          1
2      hello
dtype: object
df = pd.DataFrame([[1, np.nan, 2], [2, 3, 5], [np.nan, 4, 6]])
df
     0    1    2
0  1.0  NaN    2
1  2.0  3.0    5
2  NaN  4.0    6

df.dropna()
     0    1    2
1  2.0  3.0    5
```

```
df.dropna(axis='columns’)
     2
0    2
1    5
2    6
df.dropna(axis='columns’)
     2
0    2
1    5
2    6
df.dropna(axis='columns', how='all’)
       0     1      2
0    1.0   NaN      2
1    2.0   3.0      5
2    NaN   4.0      6
```

Filling null values

```
data = pd.Series([1, np.nan, 2, None, 3], index=list('abcde'))
Data
a     1.0
b    NaN
c    2.0
d   NaN
e    3.0
dtype: float64
data.fillna(0)
a    1.0
b    0.0
c    2.0
d    0.0
e     3.0
```

forward-fill

```
data.fillna(method='ffill')
a       1.0
b       1.0
c       2.0
d       2.0
e       3.0
dtype: float64
```

back-fill

```
data.fillna(method='bfill')
a       1.0
b       2.0
c       2.0
d       3.0
e       3.0
dtype: float64
```

HIERARCHICAL INDEXING

Hierarchical indexing (also known as *multi-indexing*) to incorporate multiple index *levels* within a single index. It is a higher-dimensional data can be compactly represented Within the familiar one-dimensional Series and two-dimensional DataFrame objects.

MultiIndex as extra dimension

```
pop_df = pop.unstack()
pop_df
```

	2000	2010
California	33871648	37253956
New York	18976457	19378102
Texas	20851820	25145561

```
pop_df.stack()
```

```
Out[9]:  California  2000    33871648
                     2010    37253956
         New York    2000    18976457
                     2010    19378102
         Texas       2000    20851820
                     2010    25145561
         dtype: int64
```

Methods of MultiIndex Creation

```
In[12]: df = pd.DataFrame(np.random.rand(4, 2),
                          index=[['a', 'a', 'b', 'b'], [1, 2, 1, 2]],
                          columns=['data1', 'data2'])
        df

Out[12]:         data1     data2
         a 1  0.554233  0.356072
           2  0.925244  0.219474
         b 1  0.441759  0.610054
           2  0.171495  0.886688
```

Similarly, if you pass a dictionary with appropriate tuples as keys, Pandas will automatically recognize this and use a `MultiIndex` by default:

```
In[13]: data = {('California', 2000): 33871648,
                ('California', 2010): 37253956,
                ('Texas', 2000): 20851820,
                ('Texas', 2010): 25145561,
                ('New York', 2000): 18976457,
                ('New York', 2010): 19378102}
        pd.Series(data)

Out[13]: California  2000    33871648
                     2010    37253956
         New York    2000    18976457
                     2010    19378102
         Texas       2000    20851820
                     2010    25145561
         dtype: int64
```

Explicit MultiIndex constructors

```
In[14]: pd.MultiIndex.from_arrays([['a', 'a', 'b', 'b'], [1, 2, 1, 2]])

Out[14]: MultiIndex(levels=[['a', 'b'], [1, 2]],
                    labels=[[0, 0, 1, 1], [0, 1, 0, 1]])
```

```
In[15]: pd.MultiIndex.from_tuples([('a', 1), ('a', 2), ('b', 1), ('b', 2)])

Out[15]: MultiIndex(levels=[['a', 'b'], [1, 2]],
                    labels=[[0, 0, 1, 1], [0, 1, 0, 1]])
```

```
In[16]: pd.MultiIndex.from_product([['a', 'b'], [1, 2]])

Out[16]: MultiIndex(levels=[['a', 'b'], [1, 2]],
                    labels=[[0, 0, 1, 1], [0, 1, 0, 1]])
```

MultiIndex level names

```
In[18]: pop.index.names = ['state', 'year']
        pop

Out[18]: state       year
         California  2000    33871648
                     2010    37253956
         New York    2000    18976457
                     2010    19378102
         Texas       2000    20851820
                     2010    25145561
         dtype: int64
```

MultiIndex for columns

```
In[19]:
# hierarchical indices and columns
index = pd.MultiIndex.from_product([[2013, 2014], [1, 2]],
                                   names=['year', 'visit'])
columns = pd.MultiIndex.from_product([['Bob', 'Guido', 'Sue'], ['HR', 'Temp']],
                                     names=['subject', 'type'])

# mock some data
data = np.round(np.random.randn(4, 6), 1)
data[:, ::2] *= 10
data += 37

# create the DataFrame
health_data = pd.DataFrame(data, index=index, columns=columns)
health_data
```

```
Out[19]: subject      Bob        Guido        Sue
         type          HR  Temp    HR  Temp    HR  Temp
         year visit
         2013 1      31.0  38.7  32.0  36.7  35.0  37.2
              2      44.0  37.7  50.0  35.0  29.0  36.7
         2014 1      30.0  37.4  39.0  37.8  61.0  36.9
              2      47.0  37.8  48.0  37.3  51.0  36.5
```

Indexing and Slicing a MultiIndex

Multiply indexed Series

Consider the multiply indexed `Series` of state populations we saw earlier:

```
In[21]: pop

Out[21]: state       year
         California  2000    33871648
                     2010    37253956
         New York    2000    18976457
                     2010    19378102
         Texas       2000    20851820
                     2010    25145561
         dtype: int64
```

We can access single elements by indexing with multiple terms:

```
In[22]: pop['California', 2000]

Out[22]: 33871648
```

Multiply indexed DataFrames

```
In[28]: health_data

Out[28]: subject      Bob        Guido         Sue
         type          HR  Temp    HR  Temp    HR  Temp
         year visit
         2013 1      31.0  38.7  32.0  36.7  35.0  37.2
              2      44.0  37.7  50.0  35.0  29.0  36.7
         2014 1      30.0  37.4  39.0  37.8  61.0  36.9
              2      47.0  37.8  48.0  37.3  51.0  36.5
```

```
In[29]: health_data['Guido', 'HR']

Out[29]: year  visit
         2013  1        32.0
               2        50.0
         2014  1        39.0
               2        48.0
         Name: (Guido, HR), dtype: float64
```

Data Aggregations on Multi-Indices

```
In[43]: health_data
```

```
Out[43]:  subject      Bob        Guido        Sue
          type          HR  Temp    HR  Temp    HR  Temp
          year visit
          2013 1      31.0  38.7  32.0  36.7  35.0  37.2
               2      44.0  37.7  50.0  35.0  29.0  36.7
          2014 1      30.0  37.4  39.0  37.8  61.0  36.9
               2      47.0  37.8  48.0  37.3  51.0  36.5
```

```
In[44]: data_mean = health_data.mean(level='year')
        data_mean
```

```
Out[44]: subject    Bob        Guido         Sue
         type        HR  Temp    HR   Temp    HR   Temp
         year
         2013      37.5  38.2  41.0  35.85  32.0  36.95
         2014      38.5  37.6  43.5  37.55  56.0  36.70
```

COMBINING DATASETS : Concat & Append

`pd.concat()` can be used for a simple concatenation of `Series` or `DataFrame` objects, just as `np.concatenate()` can be used for simple concatenations of arrays:

```
In[6]: ser1 = pd.Series(['A', 'B', 'C'], index=[1, 2, 3])
       ser2 = pd.Series(['D', 'E', 'F'], index=[4, 5, 6])
       pd.concat([ser1, ser2])

Out[6]: 1     A
        2     B
        3     C
        4     D
        5     E
        6     F
        dtype: object
```

It also works to concatenate higher-dimensional objects, such as `DataFrame`s:

```
In[7]: df1 = make_df('AB', [1, 2])
       df2 = make_df('AB', [3, 4])
       print(df1); print(df2); print(pd.concat([df1, df2]))

df1               df2               pd.concat([df1, df2])
     A    B            A    B             A    B
 1   A1   B1      3    A3   B3        1   A1   B1
 2   A2   B2      4    A4   B4        2   A2   B2
                                      3   A3   B3
                                      4   A4   B4
```

Catching the repeats as an error.

```
In[10]: try:
            pd.concat([x, y], verify_integrity=True)
        except ValueError as e:
            print("ValueError:", e)
```

```
ValueError: Indexes have overlapping values: [0, 1]
```

Concatenation with joins

```
In[13]: df5 = make_df('ABC', [1, 2])
        df6 = make_df('BCD', [3, 4])
        print(df5); print(df6); print(pd.concat([df5, df6])
```

```
df5                 df6                 pd.concat([df5, df6])
     A   B   C          B   C   D            A    B    C    D
1   A1  B1  C1      3  B3  C3  D3       1   A1   B1   C1  NaN
2   A2  B2  C2      4  B4  C4  D4       2   A2   B2   C2  NaN
                                        3  NaN   B3   C3   D3
                                        4  NaN   B4   C4   D4
```

The append() method

```
In[16]: print(df1); print(df2); print(df1.append(df2))

df1              df2              df1.append(df2)
    A   B            A   B             A   B
 1  A1  B1       3  A3  B3        1   A1  B1
 2  A2  B2       4  A4  B4        2   A2  B2
                                  3   A3  B3
                                  4   A4  B4
```

append() method in Pandas does not modify the original object—instead, it creates a new object with the combined data. It also is not a very efficient method, because it involves creation of a new index *and* data buffer. Thus, if you plan to do multiple append operations, it is generally better to build a list of DataFrames and pass them all at once to the concat() function.

<u>COMBINING DATASETS: MERGE AND JOIN</u>

Pandas implements several of these fundamental building blocks in the pd.merge()

function and the related join() method of Series and DataFrames.

Categories of Joins

The pd.merge() function implements a number of types of joins: the *one-to-one*,

many-to-one, and *many-to-many* joins.

1.One-to-one joins

```
In[2]:
df1 = pd.DataFrame({'employee': ['Bob', 'Jake', 'Lisa', 'Sue'],
                    'group': ['Accounting', 'Engineering', 'Engineering', 'HR']})
df2 = pd.DataFrame({'employee': ['Lisa', 'Bob', 'Jake', 'Sue'],
                    'hire_date': [2004, 2008, 2012, 2014]})
print(df1); print(df2)
```

```
df1                          df2
  employee        group        employee  hire_date
0      Bob   Accounting      0     Lisa       2004
1     Jake  Engineering      1      Bob       2008
2     Lisa  Engineering      2     Jake       2012
3      Sue           HR      3      Sue       2014
```

To combine this information into a single `DataFrame`, we can use the `pd.merge()` function:

```
In[3]: df3 = pd.merge(df1, df2)
       df3
```

```
Out[3]:   employee        group  hire_date
        0      Bob   Accounting       2008
        1     Jake  Engineering       2012
        2     Lisa  Engineering       2004
        3      Sue           HR       2014
```

2.Many-to-one joins

```
In[4]: df4 = pd.DataFrame({'group': ['Accounting', 'Engineering', 'HR'],
                          'supervisor': ['Carly', 'Guido', 'Steve']})
       print(df3); print(df4); print(pd.merge(df3, df4))
```

```
df3                                       df4
  employee        group  hire_date            group supervisor
0      Bob   Accounting       2008        0   Accounting      Carly
1     Jake  Engineering       2012        1  Engineering      Guido
2     Lisa  Engineering       2004        2           HR      Steve
3      Sue           HR       2014

pd.merge(df3, df4)
  employee        group  hire_date supervisor
0      Bob   Accounting       2008      Carly
1     Jake  Engineering       2012      Guido
2     Lisa  Engineering       2004      Guido
3      Sue           HR       2014      Steve
```

3.Many-to-many joins

```
In[5]: df5 = pd.DataFrame({'group': ['Accounting', 'Accounting',
                                     'Engineering', 'Engineering', 'HR', 'HR'],
                           'skills': ['math', 'spreadsheets', 'coding', 'linux',
                                      'spreadsheets', 'organization']})
print(df1); print(df5); print(pd.merge(df1, df5))
```

```
df1                                 df5
  employee        group                    group        skills
0      Bob   Accounting             0   Accounting          math
1     Jake  Engineering             1   Accounting  spreadsheets
2     Lisa  Engineering             2  Engineering        coding
3      Sue           HR             3  Engineering         linux
                                    4           HR  spreadsheets
                                    5           HR  organization


pd.merge(df1, df5)
  employee        group        skills
0      Bob   Accounting          math
1      Bob   Accounting  spreadsheets
2     Jake  Engineering        coding
3     Jake  Engineering         linux
4     Lisa  Engineering        coding
5     Lisa  Engineering         linux
6      Sue           HR  spreadsheets
7      Sue           HR  organization
```

Specification of the Merge Key

The on keyword

```
In[6]: print(df1); print(df2); print(pd.merge(df1, df2, on='employee'))
```

```
df1                              df2
  employee        group             employee  hire_date
0      Bob   Accounting          0      Lisa       2004
1     Jake  Engineering          1       Bob       2008
2     Lisa  Engineering          2      Jake       2012
3      Sue           HR          3       Sue       2014
```

```
pd.merge(df1, df2, on='employee')
  employee        group  hire_date
0      Bob   Accounting       2008
1     Jake  Engineering       2012
2     Lisa  Engineering       2004
3      Sue           HR       2014
```

The left_on and right_on keywords

```
In[7]:
df3 = pd.DataFrame({'name': ['Bob', 'Jake', 'Lisa', 'Sue'],
                    'salary': [70000, 80000, 120000, 90000]})
print(df1); print(df3);
print(pd.merge(df1, df3, left_on="employee", right_on="name"))
```

```
df1                             df3
  employee        group            name  salary
0      Bob  Accounting          0   Bob   70000
1     Jake  Engineering         1  Jake   80000
2     Lisa  Engineering         2  Lisa  120000
3      Sue           HR         3   Sue   90000


pd.merge(df1, df3, left_on="employee", right_on="name")
  employee        group  name  salary
0      Bob  Accounting    Bob   70000
1     Jake  Engineering  Jake   80000
2     Lisa  Engineering  Lisa  120000
3      Sue           HR   Sue   90000
```

AGGREGATION AND GROUPING

Computing Aggregations like sum(), mean(), median(), min(), and max()

Planets Data

```
In[2]: import seaborn as sns
       planets = sns.load_dataset('planets')
       planets.shape

Out[2]: (1035, 6)

In[3]: planets.head()

Out[3]:     method          number  orbital_period  mass   distance  year
         0  Radial Velocity  1       269.300         7.10   77.40     2006
         1  Radial Velocity  1       874.774         2.21   56.95     2008
         2  Radial Velocity  1       763.000         2.60   19.84     2011
         3  Radial Velocity  1       326.030         19.40  110.62    2007
         4  Radial Velocity  1       516.220         10.50  119.47    2009
```

Simple Aggregation in Pandas

```
In[4]: rng = np.random.RandomState(42)
       ser = pd.Series(rng.rand(5))
       ser

Out[4]: 0    0.374540
        1    0.950714
        2    0.731994
        3    0.598658
        4    0.156019
        dtype: float64

In[5]: ser.sum()

Out[5]: 2.811925491708157

In[6]: ser.mean()

Out[6]: 0.5623850983416314
```

```
In[7]: df = pd.DataFrame({'A': rng.rand(5),
                          'B': rng.rand(5)})
       df

Out[7]:           A         B
        0  0.155995  0.020584
        1  0.058084  0.969910
        2  0.866176  0.832443
        3  0.601115  0.212339
        4  0.708073  0.181825

In[8]: df.mean()

Out[8]: A    0.477888
        B    0.443420
        dtype: float64

In[9]: df.mean(axis='columns')

Out[9]: 0     0.088290
        1     0.513997
        2     0.849309
        3     0.406727
        4     0.444949
        dtype: float64
```

Listing of Pandas aggregation methods

Aggregation	Description
`count()`	Total number of items
`first()`, `last()`	First and last item
`mean()`, `median()`	Mean and median
`min()`, `max()`	Minimum and maximum
`std()`, `var()`	Standard deviation and variance
`mad()`	Mean absolute deviation
`prod()`	Product of all items
`sum()`	Sum of all items

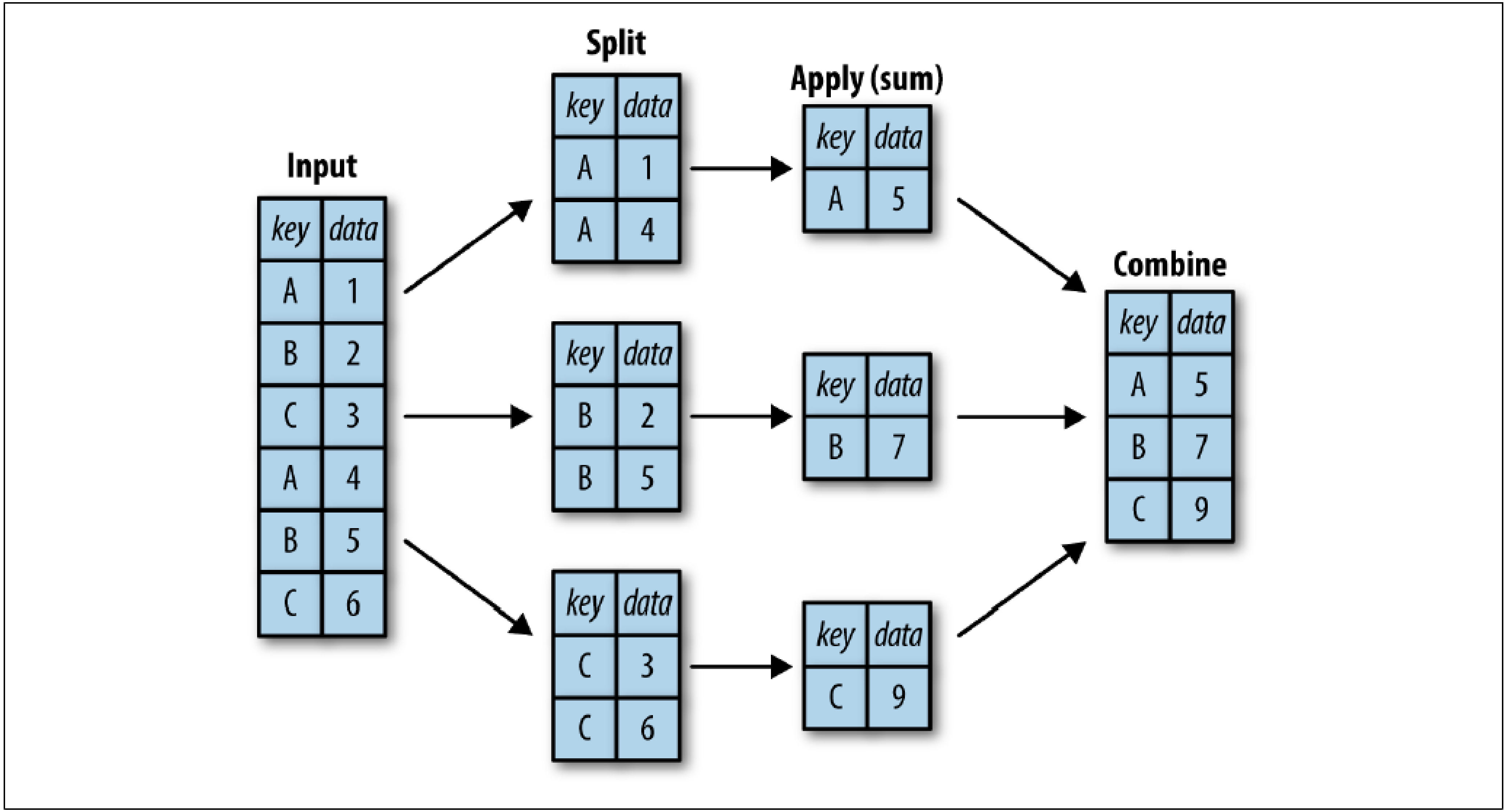
GroupBy: Split, Apply, Combine

- The *split* step involves breaking up and grouping a DataFrame depending on the value of the specified key.

- The *apply* step involves computing some function, usually an aggregate, transformation, or filtering, within the individual groups.

- The *combine* step merges the results of these operations into an output array.

A visual representation of a groupby operation



```
In[11]: df = pd.DataFrame({'key': ['A', 'B', 'C', 'A', 'B', 'C'],
                           'data': range(6)}, columns=['key', 'data'])
        df
```

```
Out[11]:   key  data
        0   A     0
        1   B     1
        2   C     2
        3   A     3
        4   B     4
        5   C     5
```

```
In[13]: df.groupby('key').sum()
```

```
Out[13]:      data
         key
         A       3
         B       5
         C       7
```

Aggregate, filter, transform, apply

```
In[19]: rng = np.random.RandomState(0)
        df = pd.DataFrame({'key': ['A', 'B', 'C', 'A', 'B', 'C'],
                           'data1': range(6),
                           'data2': rng.randint(0, 10, 6)},
                           columns = ['key', 'data1', 'data2'])
        df
```

Out[19]:

	key	data1	data2
0	A	0	5
1	B	1	0
2	C	2	3
3	A	3	3
4	B	4	7
5	C	5	9

Aggregation. We're now familiar with GroupBy aggregations with sum(), median(),and the like, but the aggregate() method allows for even more flexibility. It can take a string, a function, or a list thereof, and compute all the aggregates at once.

```
In[20]: df.groupby('key').aggregate(['min', np.median, max])

Out[20]:      data1              data2
              min median max     min median max
        key
        A       0    1.5   3       3    4.0   5
        B       1    2.5   4       0    3.5   7
        C       2    3.5   5       3    6.0   9
```

Filtering. A filtering operation allows you to drop data based on the group properties.

```
In[22]:
def filter_func(x):
    return x['data2'].std() > 4

print(df); print(df.groupby('key').std());
print(df.groupby('key').filter(filter_func))
```

```
df                               df.groupby('key').std()
    key   data1   data2          key       data1      data2
0    A       0       5           A      2.12132   1.414214
1    B       1       0           B      2.12132   4.949747
2    C       2       3           C      2.12132   4.242641
3    A       3       3
4    B       4       7
5    C       5       9
```

Transformation. While aggregation must return a reduced version of the data, transformation can return some transformed version of the full data to recombine. For such a transformation, the output is the same shape as the input.

```
In[23]: df.groupby('key').transform(lambda x: x - x.mean())

Out[23]:    data1  data2
         0   -1.5    1.0
         1   -1.5   -3.5
         2   -1.5   -3.0
         3    1.5   -1.0
         4    1.5    3.5
         5    1.5    3.0
```

The apply() method. The apply() method lets you apply an arbitrary function to the group results. The function should take a DataFrame, and return either a Pandas object (e.g., DataFrame, Series) or a scalar; the combine operation will be tailored to the type of output returned.

```
In[24]: def norm_by_data2(x):
            # x is a DataFrame of group values
            x['data1'] /= x['data2'].sum()
            return x

        print(df); print(df.groupby('key').apply(norm_by_data2))
```

```
df                          df.groupby('key').apply(norm_by_data2)
  key  data1  data2            key     data1  data2
0   A      0      5         0    A  0.000000      5
1   B      1      0         1    B  0.142857      0
2   C      2      3         2    C  0.166667      3
3   A      3      3         3    A  0.375000      3
4   B      4      7         4    B  0.571429      7
5   C      5      9         5    C  0.416667      9
```

PIVOT TABLES

- A *pivot table* is a similar operation that is commonly seen in spreadsheets and other programs that operate on tabular data.

- The pivot table takes simple columnwise data as input, and groups the entries into a two-dimensional table that provides a multidimensional summarization of the data.

- The difference between pivot tables and GroupBy can sometimes cause confusion; it helps me to think of pivot tables as essentially a *multidimensional* version of GroupBy aggregation.

```
In[1]: import numpy as np
       import pandas as pd
       import seaborn as sns
       titanic = sns.load_dataset('titanic')

In[2]: titanic.head()

Out[2]:
   survived  pclass     sex   age  sibsp  parch     fare embarked  class  \\
0         0       3    male  22.0      1      0   7.2500        S  Third
1         1       1  female  38.0      1      0  71.2833        C  First
2         1       3  female  26.0      0      0   7.9250        S  Third
3         1       1  female  35.0      1      0  53.1000        S  First
4         0       3    male  35.0      0      0   8.0500        S  Third

     who adult_male deck  embark_town alive  alone
0    man       True  NaN  Southampton    no  False
1  woman      False    C    Cherbourg   yes  False
2  woman      False  NaN  Southampton   yes   True
3  woman      False    C  Southampton   yes  False
4    man       True  NaN  Southampton    no   True
```

Pivot Tables by Hand

```
In[3]: titanic.groupby('sex')[['survived']].mean()

Out[3]:         survived
        sex
        female  0.742038
        male    0.188908
```

Pivot Table Syntax

```
In[5]: titanic.pivot_table('survived', index='sex', columns='class')

Out[5]: class      First    Second     Third
        sex
        female  0.968085  0.921053  0.500000
        male    0.368852  0.157407  0.135447
```

Multilevel pivot tables

```
In[6]: age = pd.cut(titanic['age'], [0, 18, 80])
       titanic.pivot_table('survived', ['sex', age], 'class')

Out[6]:   class                 First    Second     Third
          sex    age
          female (0, 18]     0.909091  1.000000  0.511628
                 (18, 80]    0.972973  0.900000  0.423729
          male   (0, 18]     0.800000  0.600000  0.215686
                 (18, 80]    0.375000  0.071429  0.133663
```

VECTORIZED STRING OPERATIONS

- A comprehensive set of *vectorized string operations* that become an essential piece of the type of munging required when one is working with (read: cleaning up) real-world data.
- This *vectorization* of operations simplifies the syntax of operating on arrays of data: we no longer have to worry about the size or shape of the array.
- Pandas includes features to address both this need for vectorized string operations and for correctly handling missing data via the str attribute of Pandas Series and Index objects containing strings.

```
In[4]: import pandas as pd
       names = pd.Series(data)
       names

Out[4]: 0    peter
        1     Paul
        2     None
        3     MARY
        4    gUIDO
        dtype: object
```

```
In[5]: names.str.capitalize()

Out[5]: 0    Peter
        1     Paul
        2     None
        3     Mary
        4    Guido
        dtype: object
```

Tables of Pandas String Methods

Methods similar to Python string methods

Nearly all Python's built-in string methods are mirrored by a Pandas vectorized string method. Here is a list of Pandas `str` methods that mirror Python string methods:

`len()`	`lower()`	`translate()`	`islower()`
`ljust()`	`upper()`	`startswith()`	`isupper()`
`rjust()`	`find()`	`endswith()`	`isnumeric()`
`center()`	`rfind()`	`isalnum()`	`isdecimal()`
`zfill()`	`index()`	`isalpha()`	`split()`
`strip()`	`rindex()`	`isdigit()`	`rsplit()`
`rstrip()`	`capitalize()`	`isspace()`	`partition()`
`lstrip()`	`swapcase()`	`istitle()`	`rpartition()`

Methods using regular expressions

Mapping between Pandas methods and functions in Python's re module

Method	Description
`match()`	Call `re.match()` on each element, returning a Boolean.
`extract()`	Call `re.match()` on each element, returning matched groups as strings.
`findall()`	Call `re.findall()` on each element.
`replace()`	Replace occurrences of pattern with some other string.
`contains()`	Call `re.search()` on each element, returning a Boolean.
`count()`	Count occurrences of pattern.
`split()`	Equivalent to `str.split()`, but accepts regexps.
`rsplit()`	Equivalent to `str.rsplit()`, but accepts regexps.

Miscellaneous methods

Other Pandas string methods

Method	Description
`get()`	Index each element
`slice()`	Slice each element
`slice_replace()`	Replace slice in each element with passed value
`cat()`	Concatenate strings
`repeat()`	Repeat values
`normalize()`	Return Unicode form of string
`pad()`	Add whitespace to left, right, or both sides of strings
`wrap()`	Split long strings into lines with length less than a given width
`join()`	Join strings in each element of the `Series` with passed separator
`get_dummies()`	Extract dummy variables as a `DataFrame`

WORKING WITH TIME SERIES

Pandas was developed in the context of financial modeling, it contains a fairly extensive set of tools for working with dates, times, and time indexed data.

Time stamps reference particular moments in time (e.g., July 4th, 2015, at 7:00 a.m.).

• *Time intervals* and *periods* reference a length of time between a particular beginning

and end point—for example, the year 2015. Periods usually reference a special case of time intervals in which each interval is of uniform length and does not overlap (e.g., 24 hour-long periods constituting days).

• *Time deltas* or *durations* reference an exact length of time (e.g., a duration of 22.56 seconds).

Dates and Times in Python

Native Python dates and times: datetime and dateutil

```
In[1]: from datetime import datetime
       datetime(year=2015, month=7, day=4)

Out[1]: datetime.datetime(2015, 7, 4, 0, 0)
```

```
In[2]: from dateutil import parser
       date = parser.parse("4th of July, 2015")
       date

Out[2]: datetime.datetime(2015, 7, 4, 0, 0)
```

printing the day of the week

```
In[3]: date.strftime('%A')

Out[3]: 'Saturday'
```

Pandas Time Series: Indexing by Time

```
In[12]: index = pd.DatetimeIndex(['2014-07-04', '2014-08-04',
                                  '2015-07-04', '2015-08-04'])
        data = pd.Series([0, 1, 2, 3], index=index)
        data

Out[12]: 2014-07-04    0
         2014-08-04    1
         2015-07-04    2
         2015-08-04    3
         dtype: int64
```

Pandas Time Series Data Structures

- For *time stamps*, Pandas provides the Timestamp type. As mentioned before, it is essentially a replacement for Python's native datetime, but is based on the more efficient numpy.datetime64 data type. The associated index structure is DatetimeIndex.

- For *time periods*, Pandas provides the Period type. This encodes a fixed frequency interval based on numpy.datetime64. The associated index structure is PeriodIndex.

- For *time deltas* or *durations*, Pandas provides the Timedelta type. Timedelta is a more efficient replacement for Python's native datetime.timedelta type, and is based on numpy.timedelta64. The associated index structure is TimedeltaIndex.

```
In[15]: dates = pd.to_datetime([datetime(2015, 7, 3), '4th of July, 2015',
                               '2015-Jul-6', '07-07-2015', '20150708'])
        dates

Out[15]: DatetimeIndex(['2015-07-03', '2015-07-04', '2015-07-06', '2015-07-07',
                        '2015-07-08'],
                       dtype='datetime64[ns]', freq=None)
```

```
In[16]: dates.to_period('D')

Out[16]: PeriodIndex(['2015-07-03', '2015-07-04', '2015-07-06', '2015-07-07',
                      '2015-07-08'],
                     dtype='int64', freq='D')
```

```
In[17]: dates - dates[0]

Out[17]:
TimedeltaIndex(['0 days', '1 days', '3 days', '4 days', '5 days'],
               dtype='timedelta64[ns]', freq=None)
```

FREQUENCIES AND OFFSETS

Code	Description	Code	Description
D	Calendar day	B	Business day
W	Weekly		
M	Month end	BM	Business month end
Q	Quarter end	BQ	Business quarter end
A	Year end	BA	Business year end
H	Hours	BH	Business hours
T	Minutes		
S	Seconds		
L	Milliseonds		
U	Microseconds		
N	Nanoseconds		

Listing of start-indexed frequency codes

Code	Description
MS	Month start
BMS	Business month start
QS	Quarter start
BQS	Business quarter start
AS	Year start
BAS	Business year start

MOTIVATING QUERY() AND EVAL():

pandas.eval() for Efficient Operations

The eval() function in Pandas uses string expressions to efficiently compute operations using DataFrames.

```
In[6]: import pandas as pd
       nrows, ncols = 100000, 100
       rng = np.random.RandomState(42)
       df1, df2, df3, df4 = (pd.DataFrame(rng.rand(nrows, ncols))
                             for i in range(4))
```

```
In[7]: %timeit df1 + df2 + df3 + df4

10 loops, best of 3: 87.1 ms per loop
```

```
In[8]: %timeit pd.eval('df1 + df2 + df3 + df4')

10 loops, best of 3: 42.2 ms per loop
```

Operations supported by pd.eval()

```
In[10]: df1, df2, df3, df4, df5 = (pd.DataFrame(rng.randint(0, 1000, (100, 3)))
                                   for i in range(5))
```

Arithmetic operators. pd.eval() supports all arithmetic operators

```
In[11]: result1 = -df1 * df2 / (df3 + df4) - df5
        result2 = pd.eval('-df1 * df2 / (df3 + df4) - df5')
        np.allclose(result1, result2)

Out[11]: True
```

Comparison operators. `pd.eval()` supports all comparison operators, including chained expressions:

```
In[12]: result1 = (df1 < df2) & (df2 <= df3) & (df3 != df4)
        result2 = pd.eval('df1 < df2 <= df3 != df4')
        np.allclose(result1, result2)

Out[12]: True
```

Bitwise operators. `pd.eval()` supports the & and | bitwise operators:

```
In[13]: result1 = (df1 < 0.5) & (df2 < 0.5) | (df3 < df4)
        result2 = pd.eval('(df1 < 0.5) & (df2 < 0.5) | (df3 < df4)')
        np.allclose(result1, result2)

Out[13]: True
```

Object attributes and indices. `pd.eval()` supports access to object attributes via the `obj.attr` syntax, and indexes via the `obj[index]` syntax:

```
In[15]: result1 = df2.T[0] + df3.iloc[1]
        result2 = pd.eval('df2.T[0] + df3.iloc[1]')
        np.allclose(result1, result2)

Out[15]: True
```

DataFrame.eval() for Column-Wise Operations

Assignment in DataFrame.eval()

```
In[19]: df.head()
```

```
Out[19]:          A         B         C
         0  0.375506  0.406939  0.069938
         1  0.069087  0.235615  0.154374
         2  0.677945  0.433839  0.652324
         3  0.264038  0.808055  0.347197
         4  0.589161  0.252418  0.557789
```

We can use `df.eval()` to create a new column `'D'` and assign to it a value computed from the other columns:

```
In[20]: df.eval('D = (A + B) / C', inplace=True)
        df.head()
```

```
Out[20]:          A         B         C          D
         0  0.375506  0.406939  0.069938  11.187620
         1  0.069087  0.235615  0.154374   1.973796
         2  0.677945  0.433839  0.652324   1.704344
         3  0.264038  0.808055  0.347197   3.087857
         4  0.589161  0.252418  0.557789   1.508776
```

Local variables in DataFrame.eval()

The `DataFrame.eval()` method supports an additional syntax that lets it work with local Python variables. Consider the following:

```
In[22]: column_mean = df.mean(1)
        result1 = df['A'] + column_mean
        result2 = df.eval('A + @column_mean')
        np.allclose(result1, result2)

Out[22]: True
```

DataFrame.query() Method

The DataFrame has another method based on evaluated strings, called the query() method

```
In[23]: result1 = df[(df.A < 0.5) & (df.B < 0.5)]
        result2 = pd.eval('df[(df.A < 0.5) & (df.B < 0.5)]')
        np.allclose(result1, result2)

Out[23]: True
```

```
In[24]: result2 = df.query('A < 0.5 and B < 0.5')
        np.allclose(result1, result2)

Out[24]: True
```