Getting Started with pandas

Part 5

Essential Functionality

Part 3

Arithmetic methods with fill values

- In arithmetic operations between differently indexed objects, you might want to fill with a special value, like `0`, when an axis label is found in one object but not the other.

In [134]:
```
df1 = pd.DataFrame(np.arange(12.).reshape((3, 4)),
                   columns=list('abcd'))
df2 = pd.DataFrame(np.arange(20.).reshape((4, 5)),
                   columns=list('abcde'))
```

In [135]:
```
df2.loc[1, 'b'] = np.nan
```

In [136]:
```
df1
```

Out[136]:

	a	b	c	d
0	0.0	1.0	2.0	3.0
1	4.0	5.0	6.0	7.0
2	8.0	9.0	10.0	11.0

In [137]:
```
df2
```

Out[137]:

	a	b	c	d	e
0	0.0	1.0	2.0	3.0	4.0
1	5.0	NaN	7.0	8.0	9.0
2	10.0	11.0	12.0	13.0	14.0
3	15.0	16.0	17.0	18.0	19.0

In [138]:
```
df1 + df2
```

Out[138]:

	a	b	c	d	e
0	0.0	2.0	4.0	6.0	NaN
1	9.0	NaN	13.0	15.0	NaN
2	18.0	20.0	22.0	24.0	NaN
3	NaN	NaN	NaN	NaN	NaN

- Using the `add` method on `df1`, I pass `df2` and an argument to `fill_value`:

```
In [139]: df1.add(df2, fill_value=0)
```

Out[139]:

	a	b	c	d	e
0	0.0	2.0	4.0	6.0	4.0
1	9.0	5.0	13.0	15.0	9.0
2	18.0	20.0	22.0	24.0	14.0
3	15.0	16.0	17.0	18.0	19.0

- Flexible arithmetic methods

Method	Description
`add, radd`	Methods for addition (+)
`sub, rsub`	Methods for subtraction (-)
`div, rdiv`	Methods for division (/)
`floordiv, rfloordiv`	Methods for floor division (//)
`mul, rmul`	Methods for multiplication (*)
`pow, rpow`	Methods for exponentiation (**)

- Each of them has a counterpart, starting with the letter `r`, that has arguments flipped.
- So these two statements are equivalent:

```
In [140]: 1 / df1
```

Out[140]:

	a	b	c	d
0	inf	1.000000	0.500000	0.333333
1	0.250	0.200000	0.166667	0.142857
2	0.125	0.111111	0.100000	0.090909

```
In [141]: df1.rdiv(1)
```

Out[141]:

	a	b	c	d
0	inf	1.000000	0.500000	0.333333
1	0.250	0.200000	0.166667	0.142857
2	0.125	0.111111	0.100000	0.090909

- Relatedly, when reindexing a Series or DataFrame, you can also specify a different fill value:

```
In [142]: df1.reindex(columns=df2.columns, fill_value=0)
```

Out[142]:

	a	b	c	d	e
0	0.0	1.0	2.0	3.0	0
1	4.0	5.0	6.0	7.0	0
2	8.0	9.0	10.0	11.0	0

Operations between DataFrame and Series

- As with NumPy arrays of different dimensions, arithmetic between DataFrame and Series is also defined.
- First, as a motivating example, consider the difference between a two-dimensional array and one of its rows:

```
In [143]: arr = np.arange(12.).reshape((3, 4))
          arr

Out[143]: array([[ 0.,  1.,  2.,  3.],
                 [ 4.,  5.,  6.,  7.],
                 [ 8.,  9., 10., 11.]])

In [144]: arr[0]

Out[144]: array([0., 1., 2., 3.])

In [145]: arr - arr[0]

Out[145]: array([[0., 0., 0., 0.],
                 [4., 4., 4., 4.],
                 [8., 8., 8., 8.]])
```

- When we subtract `arr[0]` from `arr`, the subtraction is performed once for each row.
- This is referred to as *broadcasting*.

- Operations between a DataFrame and a Series are similar:

In [146]:
```
frame = pd.DataFrame(np.arange(12.).reshape((4, 3)),
                     columns=list('bde'),
                     index=['Utah', 'Ohio', 'Texas', 'Oregon'])
series = frame.iloc[0]
```

In [147]:
```
frame
```

Out[147]:

	b	d	e
Utah	0.0	1.0	2.0
Ohio	3.0	4.0	5.0
Texas	6.0	7.0	8.0
Oregon	9.0	10.0	11.0

In [148]:
```
series
```

Out[148]:
```
b    0.0
d    1.0
e    2.0
Name: Utah, dtype: float64
```

In [149]:
```
frame - series
```

Out[149]:

	b	d	e
Utah	0.0	0.0	0.0
Ohio	3.0	3.0	3.0
Texas	6.0	6.0	6.0
Oregon	9.0	9.0	9.0

- If an index value is not found in either the DataFrame's columns or the Series's index, the objects will be reindexed to form the union:

```
In [150]: series2 = pd.Series(range(3), index=['b', 'e', 'f'])

In [151]: frame
```

Out[151]:

	b	d	e
Utah	0.0	1.0	2.0
Ohio	3.0	4.0	5.0
Texas	6.0	7.0	8.0
Oregon	9.0	10.0	11.0

```
In [152]: frame + series2
```

Out[152]:

	b	d	e	f
Utah	0.0	NaN	3.0	NaN
Ohio	3.0	NaN	6.0	NaN
Texas	6.0	NaN	9.0	NaN
Oregon	9.0	NaN	12.0	NaN

- If you want to instead broadcast over the columns, matching on the rows, you have to use one of the arithmetic methods.

```
In [153]: series3 = frame['d']

In [154]: frame
```

Out[154]:

	b	d	e
Utah	0.0	1.0	2.0
Ohio	3.0	4.0	5.0
Texas	6.0	7.0	8.0
Oregon	9.0	10.0	11.0

```
In [155]: series3
```

```
Out[155]: Utah       1.0
          Ohio       4.0
          Texas      7.0
          Oregon    10.0
          Name: d, dtype: float64
```

```
In [156]: frame.sub(series3, axis='index')
```

Out[156]:

	b	d	e
Utah	-1.0	0.0	1.0
Ohio	-1.0	0.0	1.0
Texas	-1.0	0.0	1.0
Oregon	-1.0	0.0	1.0

- The axis number that you pass is the *axis to match on*.
- In this case we mean to match on the DataFrame's row index (`axis='index' or axis=0`) and broadcast across.

Function Application and Mapping

- NumPy ufuncs (element-wise array methods) also work with pandas objects:

```
In [157]: frame = pd.DataFrame(np.random.randn(4, 3), columns=list('bde'),
                               index=['Utah', 'Ohio', 'Texas', 'Oregon'])

In [158]: frame
```

Out[158]:

	b	d	e
Utah	-0.204708	0.478943	-0.519439
Ohio	-0.555730	1.965781	1.393406
Texas	0.092908	0.281746	0.769023
Oregon	1.246435	1.007189	-1.296221

```
In [159]: np.abs(frame)
```

Out[159]:

	b	d	e
Utah	0.204708	0.478943	0.519439
Ohio	0.555730	1.965781	1.393406
Texas	0.092908	0.281746	0.769023
Oregon	1.246435	1.007189	1.296221

- Another frequent operation is applying a function on one-dimensional arrays to each column or row.
- DataFrame's apply method does exactly this:

```
In [160]: f = lambda x: x.max() - x.min()

In [161]: frame.apply(f)

Out[161]: b    1.802165
          d    1.684034
          e    2.689627
          dtype: float64
```

- Here the function `f`, which computes the difference between the maximum and minimum of a Series, is invoked once on each column in frame.
- The result is a Series having the columns of frame as its index.

- If you pass `axis='columns'` to `apply`, the function will be invoked once per row instead:

```
In [162]: frame.apply(f, axis='columns')

Out[162]: Utah      0.998382
          Ohio      2.521511
          Texas     0.676115
          Oregon    2.542656
          dtype: float64
```

- Many of the most common array statistics (like `sum` and `mean`) are DataFrame methods, so using `apply` is not necessary.

- The function passed to `apply` need not return a scalar value; it can also return a Series with multiple values:

```
In [163]: def f(x):
              return pd.Series([x.min(), x.max()], index=['min', 'max'])

In [164]: frame.apply(f)
```

Out[164]:

	b	d	e
min	-0.555730	0.281746	-1.296221
max	1.246435	1.965781	1.393406

- Element-wise Python functions can be used, too.
- Suppose you wanted to compute a formatted string from each floating-point value in frame.
- You can do this with `applymap`:

```
In [165]: format = lambda x: '%.2f' % x
          frame.applymap(format)
```

Out[165]:

	b	d	e
Utah	-0.20	0.48	-0.52
Ohio	-0.56	1.97	1.39
Texas	0.09	0.28	0.77
Oregon	1.25	1.01	-1.30

- The reason for the name `applymap` is that Series has a `map` method for applying an element-wise function:

```
In [166]: frame['e'].map(format)

Out[166]: Utah      -0.52
          Ohio       1.39
          Texas      0.77
          Oregon    -1.30
          Name: e, dtype: object
```