Algorithms and advanced Python programming for IE

Content

- Pandas – remaining topics

Mathematical Operations (subtraction)

```
>>> s = pd.Series([1, 2, 3, 4, 5], index = dates); s

>>> s.value_counts() # count values
1    4
0    2
dtype: int64
```

```
2013-01-01    1
2013-01-02    1
2013-01-03    1
2013-01-04    1
2013-01-05    1
2013-01-06    1
Freq: D, dtype: int64
```

```
>>> df3.sub(s, axis='index') # series is broadcasted
                   A         B         C         D
2013-01-01 -1.323716 -1.503347  0.337389 -0.353994
2013-01-02 -1.581779 -2.504592 -0.862944 -0.341277
2013-01-03 -0.837001  0.337054 -0.971274 -0.518763
2013-01-04 -2.124049 -2.964276 -0.875564 -1.405461
2013-01-05 -0.451192 -0.553276 -1.658097 -1.228244
2013-01-06 -0.412865 -1.545633  1.009149 -1.754034
```

The series is broadcasted along the columns, last lines are unchanged

Apply

- Apply a function across the axes

```
>>> df
     AAA   BBB   CCC
I0    -4   -10  -100
I1    -5   -20   -50
I2    -6   -30   -30
I3    -7   -40   -50

>>> df.apply(np.cumsum)
## np.cumsum - Return the cumulative sum of the elements
along a given axis
     AAA   BBB   CCC
I0    -4   -10  -100
I1    -9   -30  -150
I2   -15   -60  -180
I3   -22  -100  -230
```

```
>>> df.apply(np.cumsum, axis=1)
     AAA   BBB   CCC
I0    -4   -14  -114
I1    -5   -25   -75
I2    -6   -36   -66
I3    -7   -47   -97
```

Passing an argument to the applied function

Merging DataFrames

Concatenate

- To concatenate to dataframes use pd.concat([df1, df2, …])

```
pd.concat(objs, axis=0, join='outer', join_axes=None,
ignore_index=False, keys=None, levels=None, names=None,
verify_integrity=False, copy=True)
```

- objs : a sequence or mapping of Series, DataFrame
- axis : {0, 1, ...}, default 0. The axis to concatenate along

- join : {'inner', 'outer'}, default 'outer'
  - How to handle indexes on other axis(es). Outer for union and inner for intersection

- ignore_index : boolean, default False
  - If True, do not use the index values on the concatenation axis; useful when concatenation axis does not have meaningful indexing information

- keys : sequence, default None. Construct hierarchical index using the passed keys as the outermost level. If multiple levels passed, should contain tuples.

Concatenate

```
frames = [df1, df2, df3]
result = pd.concat(frames)
```

df1

	A	B	C	D
0	A0	B0	C0	D0
1	A1	B1	C1	D1
2	A2	B2	C2	D2
3	A3	B3	C3	D3

df2

	A	B	C	D
4	A4	B4	C4	D4
5	A5	B5	C5	D5
6	A6	B6	C6	D6
7	A7	B7	C7	D7

df3

	A	B	C	D
8	A8	B8	C8	D8
9	A9	B9	C9	D9
10	A10	B10	C10	D10
11	A11	B11	C11	D11

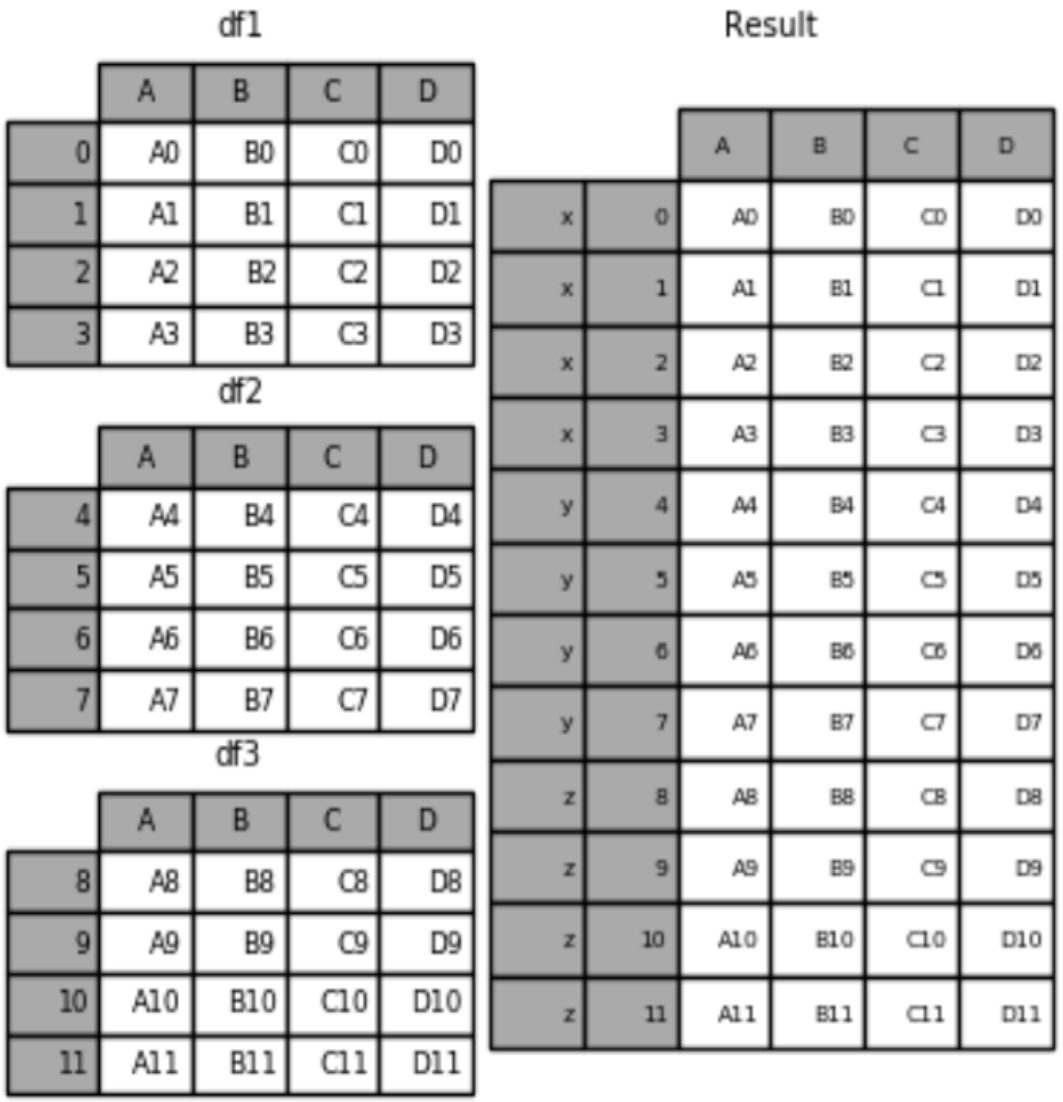
Result

	A	B	C	D
0	A0	B0	C0	D0
1	A1	B1	C1	D1
2	A2	B2	C2	D2
3	A3	B3	C3	D3
4	A4	B4	C4	D4
5	A5	B5	C5	D5
6	A6	B6	C6	D6
7	A7	B7	C7	D7
8	A8	B8	C8	D8
9	A9	B9	C9	D9
10	A10	B10	C10	D10
11	A11	B11	C11	D11

Concatenate

```
>>> result = pd.concat(frames, keys=['x','y','z'])
## the resulting object's index
## has a hierarchical index.
```

```
>>> result.loc['y']
   A   B   C   D
4  A4  B4  C4  D4
5  A5  B5  C5  D5
6  A6  B6  C6  D6
7  A7  B7  C7  D7
```



Concatenate

- When concatenating multiple DataFrames, needs to choose how to handle non- concatenated axes (e.g., columns)
  - Take the (sorted) union of them all, join='outer'
  - Take the intersection, join='inner'
  - Use a specific index (in the case of DataFrame) or indexesm, i.e. the join_axes argument

Concatenate

```
result = pd.concat([df1, df4], axis=1) ## join defaults to outer
```

df1

	A	B	C	D
0	A0	B0	C0	D0
1	A1	B1	C1	D1
2	A2	B2	C2	D2
3	A3	B3	C3	D3

df4

	B	D	F
2	B2	D2	F2
3	B3	D3	F3
6	B6	D6	F6
7	B7	D7	F7

Result

	A	B	C	D	B	D	F
0	A0	B0	C0	D0	NaN	NaN	NaN
1	A1	B1	C1	D1	NaN	NaN	NaN
2	A2	B2	C2	D2	B2	D2	F2
3	A3	B3	C3	D3	B3	D3	F3
6	NaN	NaN	NaN	NaN	B6	D6	F6
7	NaN	NaN	NaN	NaN	B7	D7	F7

Concatenate

```
result = pd.concat([df1, df4], axis=1, join='inner')
```

df1

	A	B	C	D
0	A0	B0	C0	D0
1	A1	B1	C1	D1
2	A2	B2	C2	D2
3	A3	B3	C3	D3

df4

	B	D	F
2	B2	D2	F2
3	B3	D3	F3
6	B6	D6	F6
7	B7	D7	F7

Result

	A	B	C	D	B	D	F
2	A2	B2	C2	D2	B2	D2	F2
3	A3	B3	C3	D3	B3	D3	F3

Concatenate

```
result = pd.concat([df1, df4], axis=1,
join_axes=[df1.index])
```

df1

	A	B	C	D
0	A0	B0	C0	D0
1	A1	B1	C1	D1
2	A2	B2	C2	D2
3	A3	B3	C3	D3

df4

	B	D	F
2	B2	D2	F2
3	B3	D3	F3
6	B6	D6	F6
7	B7	D7	F7

Result

	A	B	C	D	B	D	F
0	A0	B0	C0	D0	NaN	NaN	NaN
1	A1	B1	C1	D1	NaN	NaN	NaN
2	A2	B2	C2	D2	B2	D2	F2
3	A3	B3	C3	D3	B3	D3	F3

Concatenate

df1

	A	B	C	D
0	A0	B0	C0	D0
1	A1	B1	C1	D1
2	A2	B2	C2	D2
3	A3	B3	C3	D3

```
>>> pd.concat([df1, df2], axis=0, join='inner')
```

	B	C	D
0	B0	C0	D0
1	B1	C1	D1
2	B2	C2	D2
3	B3	C3	D3
0	B4	C4	D4
1	B5	C5	D5
2	B6	C6	D6
6	B7	C7	D7

df2

	B	C	D	E
0	B4	C4	D4	E4
1	B5	C5	D5	E5
2	B6	C6	D6	E6
6	B7	C7	D7	E7

```
>>> pd.concat([df1, df2], axis=0, join='outer')
```

	A	B	C	D	E
0	A0	B0	C0	D0	NaN
1	A1	B1	C1	D1	NaN
2	A2	B2	C2	D2	NaN
3	A3	B3	C3	D3	NaN
0	NaN	B4	C4	D4	E4
1	NaN	B5	C5	D5	E5
2	NaN	B6	C6	D6	E6
6	NaN	B7	C7	D7	E7

Append

df

	A	B	C	D
0	A0	B0	C0	D0
1	A1	B1	C1	D1
2	A2	B2	C2	D2
3	A3	B3	C3	D3

```
df.append(pd.Series(["A4","B4","C4","D4"],index=
['A','B','C','D']), ignore_index=True)
## equiv. to df1.loc[4] = ["A4","B4","C4","D4"]
```

	A	B	C	D
0	A0	B0	C0	D0
1	A1	B1	C1	D1
2	A2	B2	C2	D2
3	A3	B3	C3	D3
4	A4	B4	C4	D4

Grouping

Grouping

- **Splitting** the data into groups based on some criteria
- **Applying** a function to each group independently
  - **Aggregation** computing a summary statistic (or statistics) about each group
    - Compute group sums or means
    - Compute group sizes / counts
  - **Transformation** perform some group-specific computations and return a like-indexed
    - Standardizing data (zscore) within group
    - Filling NAs within groups with a value derived from each group
  - **Filtration**
    - Discarding data that belongs to groups with only a few members
    - Filtering out data based on the group sum or mean
- **Combining** the results into a data structure

Credits: https://pandas.pydata.org/pandas-docs/stable/groupby.html#groupby

Grouping

- Use groupby(by, axis,…) command
  - by: mapping, function, str
  - axis: int, defaults 0

```
# group by columns then apply sum
df.groupby('A', sort=True).sum()
# syntactic sugar for df.groupby(['A']); sorts output
```

	A	B	C	D
0	foo	one	1.20287-	0.05522-
1	bar	one	1.81447-	2.395985
2	foo	two	1.018601	1.552825
3	bar	three	0.59545-	0.166599
4	foo	two	1.395433	0.047609
5	bar	two	0.39267-	0.13647-
6	foo	one	0.007207	0.56176-
7	foo	three	1.928123	1.62303-

A	C	D
bar	2.80259-	2.42611
foo	3.146492	0.63958-

```
# group by columns then apply sum
df.groupby(['A','B']).sum()
```

		C	D
A	B		
bar	one	1.81447-	2.395985
	three	0.59545-	0.166599
	two	0.39267-	0.13647-
foo	one	1.19567-	0.61698-
	three	1.928123	1.62303-
	two	2.414034	1.600434

Grouping

```
def get_index(i):
    return ”small” if i>4 else ”big”

grouped = df.groupby(get_index, axis=0)
## iterating through groups
for g, vals in grouped:
    print(g,”:\n”, vals)
```

df

	A	B	C	D
0	foo	one	0.8917-	0.89972-
1	bar	one	0.8132-	0.520269
2	foo	two	1.63014-	0.39973-
3	bar	three	0.16933-	1.23593-
4	foo	two	0.61185-	0.59609-
5	bar	two	1.69088-	0.511187
6	foo	one	0.34387-	1.2952-
7	foo	three	0.31537	1.05752-

big	:			
	A	B	C	D
0	foo	one	0.58002-	0.403615
1	bar	one	1.43939-	0.475266
2	foo	two	0.948675	0.875232
3	bar	three	0.1012-	0.210458
4	foo	two	0.26174-	0.414949
small	:			
	A	B	C	D
5	bar	two	0.07919-	0.727548
6	foo	one	0.610581	1.22265-
7	foo	three	0.01489-	0.760891

groupby – functions and attributes

- df.groupby('group_criteria').get_group('group_name') – get specific group
- df.groupby('group_criteria').groups – get groups
- len(df.groupby('group_criteria') - get number of groups
- Groupby object supports the following operations:

gb.agg	gb.boxplot	gb.cummin	gb.describe	gb.filter
gb.aggregate	gb.count	gb.cumprod	gb.dtype	gb.first
gb.apply	gb.cummax	gb.cumsum	gb.fillna	gb.gender
gb.get_group	gb.height	gb.last	gb.median	gb.ngroups
gb.groups	gb.hist	gb.max	gb.min	gb.nth
gb.head	gb.indices	gb.mean	gb.name	gb.ohlc
gb.plot	gb.rank	gb.std	gb.transform	
gb.prod	gb.resample	gb.sum	gb.var	
gb.quantile	gb.size	gb.tail	gb.weight	

Aggregation

```
grouped = df.groupby(”A”)
print(grouped.aggregate(np.sum))
```

	C	D
A		
bar	0.633776	0.4961-
foo	3.624163	1.247444

```
grouped = df.groupby([”A“,”B”])
print(grouped.aggregate(np.sum))
```

	C	D	
A	B		
bar	one	1.10561-	0.294969
	three	2.27715-	1.44949-
	two	0.100199	0.85186-
foo	one	0.545806	1.819597
	three	0.274372	0.622378
	two	0.868895	1.25204-

df

	A	B	C	D
0	foo	one	0.8917-	0.89972-
1	bar	one	0.8132-	0.520269
2	foo	two	1.63014-	0.39973-
3	bar	three	0.16933-	1.23593-
4	foo	two	0.61185-	0.59609-
5	bar	two	1.69088-	0.511187
6	foo	one	0.34387-	1.2952-
7	foo	three	0.31537	1.05752-

Aggregation

df

	A	B	C	D
0	foo	one	0.8917-	0.89972-
1	bar	one	0.8132-	0.520269
2	foo	two	1.63014-	0.39973-
3	bar	three	0.16933-	1.23593-
4	foo	two	0.61185-	0.59609-
5	bar	two	1.69088-	0.511187
6	foo	one	0.34387-	1.2952-
7	foo	three	0.31537	1.05752-

```
grouped = df.groupby(["A","B"])
print(grouped.describe())
```

	C	\							D									
	count	mean	std	min	25%	50%	75%	max	count	mean	std	min	25%	50%	75%	max		
A	B							B							B			
bar	one	1	1.502363	NaN	1.502363	1.502363	1.502363	1.502363	one	1.502363	1	0.26072-	NaN	0.26072-	0.26072-	0.26072-	one	0.26072-	0.26072-
	three	1	0.75931-	NaN	0.75931-	0.75931-	0.75931-	0.75931-	three	0.75931-	1	1.33759	NaN	1.33759	1.33759	1.33759	three	1.33759	1.33759
	two	1	1.435457	NaN	1.435457	1.435457	1.435457	1.435457	two	1.435457	1	0.950688	NaN	0.950688	0.950688	0.950688	two	0.950688	0.950688
foo	one	2	0.88625-	1.055124	1.63233-	1.25929-	0.88625-	0.5132-	one	0.14016-	2	0.464817	0.941636	0.20102-	0.131899	0.464817	one	0.797736	1.130655
	three	1	0.33747-	NaN	0.33747-	0.33747-	0.33747-	0.33747-	three	0.33747-	1	1.4954-	NaN	1.4954-	1.4954-	1.4954-	three	1.4954-	1.4954-
	two	2	0.68968-	0.398616	0.97154-	0.83061-	0.68968-	0.54875-	two	0.40782-	2	0.306183	0.671977	0.16898-	0.068604	0.306183	two	0.543763	0.781343

```
## apply multiple function at once
print(grouped['C'].agg([np.sum, np.mean, np.std]))
```

		sum	mean	std
A	B			
bar	one	0.718617	0.718617	NaN
	three	1.54694-	1.54694-	NaN
	two	0.77223-	0.77223-	NaN
foo	one	0.36141-	0.18071-	0.662254
	three	1.23003-	1.23003-	NaN
	two	1.528912	0.764456	0.043093

Transformation

- Replacing missing values with group avg

```
df = pd.DataFrame({'A': [1, 2, 3, 4, 1, 2, 3, 4, 1, 2, 3, 4]
      ,"B": [1, 2, 3, 4, 1, 2, 3, 4, np.nan, np.nan, np.nan, np.nan]})
grouped = df.groupby("A")
print(grouped.count()) # count none nan values
f = lambda x: x.fillna(x.mean())
transformed = grouped.transform(f)
df["B"] = transformed
grouped = df.groupby("A")
print(grouped.count())
```

- Other useful function: `fillna`, `ffill`, `bfill`, `shift` (of `groupby`)

df →

	A	B
0	1	1
1	2	2
2	3	3
3	4	4
4	1	1
5	2	2
6	3	3
7	4	4
8	1	NaN
9	2	NaN
10	3	NaN
11	4	NaN

A	B
1	2
2	2
3	2
4	2

count

transformed →

	B
0	1
1	2
2	3
3	4
4	1
5	2
6	3
7	4
8	1
9	2
10	3
11	4

Count(2) ↘

A	
1	3
2	3
3	3
4	3

Filtration

- The filter method returns a subset of the original object
- The argument of filter must be a function that, **applied to the group as a whole**, returns True or False

```
>>> dff = pd.DataFrame({'A': np.arange(8), 'B': list('aabbbbcc')})
>>> dff.groupby('B').filter(lambda x: len(x) > 2)
   A  B
2  2  b
3  3  b
4  4  b
5  5  b
```

dff

	A	B
0	0	a
1	1	a
2	2	b
3	3	b
4	4	b
5	5	b

Filtration

- For DataFrames with multiple columns, filters should explicitly specify a column as the filter criterion

```
>>> dff = pd.DataFrame({'A': np.arange(8), 'B': list('aabbbbcc')})
>>> dff['C'] = np.arange(8)
>>> dff.groupby('B').filter(lambda x: len(x['C']) > 2)
```

	A	B	C
2	2	b	2
3	3	b	3
4	4	b	4
5	5	b	5

dff

	A	B	C
0	0	a	0
1	1	a	1
2	2	b	2
3	3	b	3
4	4	b	4
5	5	b	5
6	6	c	6
7	7	c	7