Data Manipulation With Pandas
Hierarchical Indexing

UBCO Master of Data Science – DATA 542

Motivation

https://open.canada.ca/data/en/dataset/f140e1d0-ed60-4818-85d6-f02fcb69fda1

Take a look at the following file:

Wapusk_NP_Coastal_Marine_Snowpack_2006-2016_Data

Station location	Ecozone	Site code	Type of ecosystem	Year	Sample number	Depth (cm)
Emplacement de la station	Écozone	Code de site	Type d’écosystème	Année	Numéro d’échantillon	Profondeur (cm)
Mary Lake	Forest	MLK	forest	2006	1	56
Mary Lake	Forest	MLK	forest	2006	2	38
Mary Lake	Forest	MLK	forest	2006	3	46
Mary Lake	Forest	MLK	forest	2006	4	44
Mary Lake	Forest	MLK	forest	2006	5	41
Mary Lake	Forest	MLK	forest	2006	6	34
Mary Lake	Forest	MLK	forest	2006	7	34
Mary Lake	Forest	MLK	forest	2006	8	38
Mary Lake	Forest	MLK	forest	2006	9	38
Mary Lake	Forest	MLK	forest	2006	10	33
Mary Lake	Forest	MLK	forest	2006	11	45
Mary Lake	Forest	MLK	forest	2006	12	37
Mary Lake	Forest	MLK	forest	2006	13	47
Mary Lake	Forest	MLK	forest	2006	14	42
Mary Lake	Forest	MLK	forest	2006	15	42
Mary Lake	Forest	MLK	forest	2006	16	48
Mary Lake	Forest	MLK	forest	2006	17	56
Mary Lake	Forest	MLK	forest	2006	18	50
Mary Lake	Forest	MLK	forest	2006	19	64

Motivation cont.

Multiple sources of data

Combine or join data

Make distinct analysis levels

Hierarchical indexing

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

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

Discuss: How will you analyze this data?
e.g. calculate the population in each year, or by each state?

First solution

```
index = [('California', 2000), ('California',
2010), ('New York', 2000), ('New York', 2010),
('Texas', 2000), ('Texas', 2010)]


populations = [33871648, 37253956, 18976457,
19378102, 20851820, 25145561]

pop = pd.Series(populations, index=index)

pop
```

Discuss: 1) pros and cons of this solution

2) Select values for 2010 only

Better solution: Hierarchical indexing

```
index = pd.MultiIndex.from_tuples(index)
pop = pop.reindex(index)
```

Blank entries means the same value as above

This is an multiply indexed Series

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

MultiIndex is like an extra dimension

Discuss: Select values for 2010 only

Indexing a MultiIndex Series

Index the individual elements by multiple terms:

```
pop['California', 2000]
```

Partial indexing: Indexing on just one of the levels

The outer level:

```
pop['California']
```

The inner level: pass an empty index for the outer level:

```
pop[:,2010]
```

Slicing a MultiIndex Series

Slicing can only be done on **sorted indices**. Otherwise it will be a key error.

```
pop.loc['California':'New York’]
```

The inner level:

```
pop.loc[:, 2000]
```

Using Boolean masks:

```
pop[pop > 2200000]
```

Fancy indexing

```
pop[['California','Texas']]
```

Question

Question 1: which one selects the values for 2010 only, for all states?

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

A) `pop.loc[:,2010]`

B) `pop[:,2010]`

C) `pop.loc[2010]`

D) `pop.loc['California', 2010]`

E) A and B

Stacking and unstacking

We can have one of the indices as a column

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

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

Or convert any dataframe to multi-indexing

```
pop_df.stack()
```

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

Flexibility with hierarchical indexing

Represent data of three or more dimensions in a Series or DataFrame

You may add another column

```
pop_df = pd.DataFrame({'total': pop,
 'under18': [9267089, 9284094,4687374, 4318033,5906301,
6879014]})
```

Use functions

```
u18_percent = pop_df['under18']/pop_df['total']
u18_percent.unstack()
```

	2000	2010
California	0.273594	0.249211
New York	0.247010	0.222831
Texas	0.283251	0.273568

Create hierarchical DataFrames

1- Pass list of lists as the index or column when creating a DataFrame

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

2- Use tuples as the Keys when using Dictionaries to create a DataFrame

```
data = {('California', 2000): 33871648,
('California', 2010): 37253956, ('Texas', 2000):
20851820, ('Texas', 2010): 25145561, ('New York',
2000): 18976457, ('New York', 2010): 19378102}
```

```
pd.Series(data)
```

Create hierarchical DataFrames cont.

3- Explicitly use MultiIndex constructor

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

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

From Cartesian product of single indices

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

Naming index levels

Name the indexes in hierarchical indexing

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

This is a name and not a label. So, partial indexing is through the levels attribute of the `MultiIndex` object. For example, it is an error: `pop.loc['state']` while you can run `pop.loc['California']`

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

Try it yourself

Columns can have multiple levels of indexing. Try the following code and discuss slicing or sub-setting the DataFrame

```
# 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
```

Question: Show the results for Bob

Indexing a multiply index DataFrame

The indexing operations on Multiply index Series applies on the columns

TRY IT

```
health_data[‘Bob’]
health_data[‘Bob’, ‘HR’]
health_data.iloc[:2, :2]
```

Each individual index in loc or iloc can be passed a tuple of multiple indices:

```
health_data.loc[:,('Sue', 'Temp’)]
health_data.loc[(2013, 1),('Sue', 'Temp’)]
```

Do not create slice within a tuple: Syntax errors:

```
health_data.loc[(:, 1), (:, 'HR')]
```

Indexing a multiply index DataFrame cont.

Syntax erros: `health_data.loc[(:, 1), (:, 'HR’)]`

Use python IndexSlice object:

TRY IT

```
idx = pd.IndexSlice
health_data.loc[idx[:, 1], idx[:, 'HR']]
```

	subject	Bob	Guido	Sue
	type	HR	HR	HR
year	visit			
2013	1	40.0	20.0	39.0
2014	1	25.0	51.0	49.0

Sorting hierarchical indexing DataFrame

Sort the data values based on each level of the index labels or column labels using

```
 data_frame.sort_index(level = index_level,
axis= axis)
```

Level starts at 0 indicating the outer index level

Use level to indicate explicitly which level of the indexing to sort

By default, it sorts the indexes.

Use axis to apply sorting on columns.

Sorting hierarchical indexing DataFrame cont.

```
frame = pd.DataFrame(
 np.arange(18).reshape((6, 3)),
 index=[
  ['a', 'a', 'c′, 'c′, 'b', 'b′],
  [1, 2, 2,1,1, 2]],
 columns=[
 ['Ohio', 'Ohio','Colorado'],
 ['Green', 'Red', 'Green']])
```

		Ohio		Colorado
		Green	Red	Green
a	1	0	1	2
	2	3	4	5
c	2	6	7	8
	1	9	10	11
b	1	12	13	14
	2	15	16	17

Partial slicing returns key error: `data['a':'b']`

Sorting hierarchical indexing DataFrame cont.

```
Frame.sort_index()
Frame.sort_index(level=0)
```

		Ohio		Colorado
		Green	Red	Green
a	1	0	1	2
	2	3	4	5
b	1	12	13	14
	2	15	16	17
c	1	9	10	11
	2	6	7	8

```
Frame.sort_index(level=1)
```

		Ohio		Colorado
		Green	Red	Green
a	1	0	1	2
b	1	12	13	14
c	1	9	10	11
a	2	3	4	5
b	2	15	16	17
c	2	6	7	8

Sorting hierarchical indexing DataFrame cont.

`Frame.sort_index(axis = 1)`

		Colorado	Ohio	
		Green	Green	Red
a	1	2	0	1
	2	5	3	4
c	2	8	6	7
	1	11	9	10
b	1	14	12	13
	2	17	15	16

Use assignment or `inplace = True` to modify the DataFrame not its copy

Reorder hierarchical indexing DataFrame cont.

```
frame.swaplevel()
frame.swaplevel(0,1)
frame.swaplevel('key1','key2')
```

		Colorado	Ohio	
		Green	Green	Red
key2	key1			
1	a	2	0	1
2	a	5	3	4
1	b	14	12	13
2	b	17	15	16
1	c	11	9	10
2	c	8	6	7

Unstacking multiply index DataFrames

Unstack the hierarchical index DataFrames (reduce dimension) using

```
data_frame.unstack(level = level)
```

Recover using stack()

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

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

```
pop.unstack(level = 1)
```

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

```
pop.unstack(level = 0)
```

Rearranging DataFrames

Flattening the hierarchical index DataFrame

Use index as the columns

```
pop.reset_index(name =
'population')
```

	state	year	population
0	California	2000	33871648
1	California	2010	37253956
2	New York	2000	18976457
3	New York	2010	19378102
4	Texas	2000	20851820
5	Texas	2010	25145561

```
flat_pop.set_index(['year',
'state'])
```

		population
year	**state**	
2000	**California**	33871648
2010	**California**	37253956
2000	**New York**	18976457
2010	**New York**	19378102
2000	**Texas**	20851820
2010	**Texas**	25145561

Rearranging DataFrames cont.

Use columns as index

1 `frame.set_index([‘c’,’d’])`

2 `frame.set_index(['c', 'd'], drop=False)`

	a	b	c	d
0	0	7	one	0
1	1	6	one	1
2	2	5	one	2
3	3	4	two	0
4	4	3	two	1
5	5	2	two	2
6	6	1	two	3

1

c	d	a	b
one	**0**	0	7
	1	1	6
	2	2	5
two	**0**	3	4
	1	4	3
	2	5	2
	3	6	1

2

c	d	a	b	c	d
one	**0**	0	7	one	0
	1	1	6	one	1
	2	2	5	one	2
two	**0**	3	4	two	0
	1	4	3	two	1
	2	5	2	two	2
	3	6	1	two	3

Summary statistics in each level: aggregations

Apply built-in aggregation functions such as sum(), mean(), max() on a particular level and on a specified axis. Axis is required for operations on columns.

```
data_frame.method(level =level_name, axis = axis)
```

	subject	Bob		Guido		Sue	
	type	HR	Temp	HR	Temp	HR	Temp
year	visit						
2013	1	114.0	111.0	94.0	111.9	113.0	113.3
	2	118.0	111.2	122.0	110.3	109.0	110.9
2014	1	99.0	111.6	125.0	111.7	123.0	111.4
	2	114.0	111.8	106.0	111.7	91.0	111.3

```
health_data.columns.names
```

```
FrozenList(['subject', 'type'])
```

```
health_data.columns.labels
```

```
FrozenList([[0, 0, 1, 1, 2, 2], [0, 1, 0, 1, 0, 1]])
```

```
health_data.columns.levels
```

```
FrozenList([['Bob', 'Guido', 'Sue'], ['HR', 'Temp']])
```

Summary statistics in each level: aggregations cont.

```
health_data.mean(level='year’)
```

subject	Bob		Guido		Sue	
type	HR	Temp	HR	Temp	HR	Temp
year						
2013	116.0	111.1	108.0	111.1	111.0	112.10
2014	106.5	111.7	115.5	111.7	107.0	111.35

```
health_data.sum(axis = 1,
                level='type')
```

	type	HR	Temp
year	visit		
2013	1	321.0	336.2
	2	349.0	332.4
2014	1	347.0	334.7
	2	311.0	334.8

Learning outcomes

At the end of this lecture you should be able to:

Create MultiIndex objects

Perform indexing and slicing on multiply indexed data

Compute statistics across multiply indexed data

Convert between simple and hierarchically indexed representations of data

Combining and Merging data With Pandas

UBCO Master of Data Science – DATA 542

Combining and merging data

Various sources of data

Merge with:

1) `pandas.merge`: connects rows in DataFrames based on one or more keys. Similar to database *join* operations.

2) `pandas.concat`: concatenates along an axis.

1) `pandas.merge`

`pandas.merge()` implements three types of join:

One-to-one

Many-to-one

Many-to-many

`pandas.merge(df_1, df_2, OPTIONS)`

`OPTIONS:`

- `On:` Column names to join on. Must be found in both dataframes.
- `right_on:` Columns in right DataFrame to use as join keys.
- `left_on:` Columns in left DataFrame to use as join keys.
- `left_index:` Use row index in left as its join key (or keys, if a MultiIndex).
- `right_index:` Use row index in right as its join key (or keys, if a MultiIndex).
- `how:` One of 'inner', 'outer', 'left', or 'right'; defaults to 'inner'.
- `Copy:` If False, avoid copying data into resulting data structure in some exceptional cases; by default always copies.

Example: one-to-one join

Census dataset – BC - 2016

	census_year	geo_name	dissemination_profiles	sex_male
0	2016	British Columbia / Colombie-Britannique	Total - Age groups and average age of the popu...	2278245
1	2016	British Columbia / Colombie-Britannique	0 to 4 years	113355
2	2016	British Columbia / Colombie-Britannique	5 to 9 years	122070
3	2016	British Columbia / Colombie-Britannique	10 to 14 years	119975

	census_year	geo_name	dissemination_profiles	sex_female
0	2016	British Columbia / Colombie-Britannique	Total - Age groups and average age of the popu...	2369810
1	2016	British Columbia / Colombie-Britannique	0 to 4 years	107270
2	2016	British Columbia / Colombie-Britannique	5 to 9 years	114830
3	2016	British Columbia / Colombie-Britannique	10 to 14 years	113885

Merge

```
pandas.merge(df_male, df_female)
```

	census_year	geo_name	dissemination_profiles	sex_male	sex_female
0	2016	British Columbia / Colombie-Britannique	Total - Age groups and average age of the popu...	2278245	2369810
1	2016	British Columbia / Colombie-Britannique	0 to 4 years	113355	107270
2	2016	British Columbia / Colombie-Britannique	5 to 9 years	122070	114830
3	2016	British Columbia / Colombie-Britannique	10 to 14 years	119975	113885
4	2016	British Columbia / Colombie-Britannique	15 to 19 years	133000	125980
5	2016	British Columbia / Colombie-Britannique	20 to 24 years	147615	139945

Joins on the shared columns

Merge cont.

```
pandas.merge(df_male, df_female, on =
['census_year','dissemination_profiles'])
```

	census_year	geo_name_x	dissemination_profiles	sex_male	geo_name_y	sex_female
0	2016	British Columbia / Colombie-Britannique	Total - Age groups and average age of the popu...	2278245	British Columbia / Colombie-Britannique	2369810
1	2016	British Columbia / Colombie-Britannique	0 to 4 years	113355	British Columbia / Colombie-Britannique	107270
2	2016	British Columbia / Colombie-Britannique	5 to 9 years	122070	British Columbia / Colombie-Britannique	114830
3	2016	British Columbia / Colombie-Britannique	10 to 14 years	119975	British Columbia / Colombie-Britannique	113885

Joins on the specified keys. If there are shared keys, it prefixes with _x and _y.

Modify names using `suffixes=(required_name)` attribute: `suffixes=('_m', '_f')`

Inner join

Merging on the intersection of two DataFrames

```
pd.merge(df6, df7, how='inner')
```

`df6`

	name	food
0	Peter	fish
1	Paul	beans
2	Mary	bread

`df7`

	name	drink
0	Mary	wine
1	Joseph	beer

`pd.merge(df6, df7)`

	name	food	drink
0	Mary	bread	wine

Outer join

An *outer join* returns a join over the union of the input columns, and fills in all missing values with NANs

```
pd.merge(df6, df7, how='outer’)
```

Discuss the results

Left/right join

An *left join* returns a join preserving the entries in the left dataset. It fills the missing values with NaNs. The right join is similar, keeping the entries from the right DataFrame.

```
pd.merge(df6, df7, how='left')
```

Discuss the results

Joining on hierarchical DataFrames

Use multiple keys as the join keys

```
pd.merge(left_df, right_df, left_on=['key1',
'key2'], right_index=True)
```

`left_df`

	key1	key2	data
0	Ohio	2000	0.0
1	Ohio	2001	1.0
2	Ohio	2002	2.0
3	Nevada	2001	3.0
4	Nevada	2002	4.0

`right_df`

		event1	event2
Nevada	**2001**	0	1
	2000	2	3
Ohio	**2000**	4	5
	2000	6	7
	2001	8	9
	2002	10	11

It is also possible to use indices of both sides

Discuss the results

Question

Question 1: What is the result of the following code?

```
pd.merge(left_df, right_df, left_on=['key1',
'key2'], right_index=True, how ='outer')
```

left_df:

	key1	key2	data
0	Ohio	2000	0.0
1	Ohio	2001	1.0
2	Ohio	2002	2.0
3	Nevada	2001	3.0
4	Nevada	2002	4.0

right_df:

		event1	event2
Nevada	**2001**	0	1
	2000	2	3
Ohio	**2000**	4	5
	2000	6	7
	2001	8	9
	2002	10	11

A)

	key1	key2	data	event1	event2
0	Ohio	2000	0.0	4.0	5.0
0	Ohio	2000	0.0	6.0	7.0
1	Ohio	2001	1.0	8.0	9.0
2	Ohio	2002	2.0	10.0	11.0
3	Nevada	2001	3.0	0.0	1.0
4	Nevada	2002	4.0	NaN	NaN
4	Nevada	2000	NaN	2.0	3.0

B)

	key1	key2	data	event1	event2
0	Ohio	2000	0.0	4	5
0	Ohio	2000	0.0	6	7
1	Ohio	2001	1.0	8	9
2	Ohio	2002	2.0	10	11
3	Nevada	2001	3.0	0	1

C)

	key1	key2	data	event1	event2
0	Ohio	2000	0.0	4.0	5.0
0	Ohio	2000	0.0	6.0	7.0
1	Ohio	2001	1.0	8.0	9.0
2	Ohio	2002	2.0	10.0	11.0
3	Nevada	2001	3.0	0.0	1.0
4	Nevada	2002	4.0	NaN	NaN

Join attribute

`join` is an attribute of DataFrame objects with operations similar to `merge()` to merge on indices.

```
df1.join([df_list], on=key, how=how)
```

- You can pass a list of DataFrames or a single DataFrame
- `On:` Column names to join on. Must be found in both dataframes.
- `how:` One of 'inner', 'outer', 'left', or 'right'; defaults to 'inner’.

join defaults to left join

Concatenation along axes

Think about:

1) "If the objects are indexed differently on the other axes, should we combine the distinct elements in these axes or use only the shared values (the intersection)?"

2) "Do the concatenated chunks of data need to be identifiable in the resulting object?"

3) "Does the "concatenation axis" contain data that needs to be preserved? In many cases, the default integer labels in a DataFrame are best discarded during concatenation."

Python for Data Analysis, 2nd edition, By: Wes McKinney

Concatenation along axes cont.

Use `pd.concat()` with the following attributes:

- `obj`: You can pass a list of Pandas objects
- `axis`: The axis along which the join operates
- `keys`: Values to associate with objects being concatenated, forming a hierarchical index along the concatenation axis.
- `how`: One of 'inner', 'outer', 'left', or 'right'; defaults to 'inner’.
- `verify_integrity`: Check new axis in concatenated object for duplicates and raise exception if so; by default (False) allows duplicates.
- `ignore_index`: Do not preserve indexes along concatenation axis, instead producing a new range(total_length) index
- `names`: Names for created hierarchical levels if keys and/or levels passed
- `join_axes`: Specific indexes to use for the other *n*–1 axes instead of performing union/intersection logic

Concatenation example

```
s1 = pd.Series([0, 1], index=['a', 'b'])
s2 = pd.Series([2, 3, 4], index=['c','d', 'e'])
s3 = pd.Series([5, 6], index=['f', 'g'])
```

```
pd.concat([s1, s2, s3],
axis=1, sort=False)
```

	0	1	2
a	0.0	NaN	NaN
b	1.0	NaN	NaN
c	NaN	2.0	NaN
d	NaN	3.0	NaN
e	NaN	4.0	NaN
f	NaN	NaN	5.0
g	NaN	NaN	6.0

```
pd.concat([s1, s2, s3])
```

a	0
b	1
c	2
d	3
e	4
f	5
g	6

Question

Question 1: What is the result of the following code?

```
s1 = pd.Series([0, 1], index=['a', 'b'])

s4 = pd.Series([0, 1, 5, 6],
index=['a','b','f', 'g'])

pd.concat([s1, s4], axis=1, sort=True)
```

A)

	0	1
a	0.0	0
b	1.0	1
f	NaN	5
g	NaN	6

B)

	0	1
a	0	0
b	1	1

C)

```
a        0
b        1
a        0
b        1
f        5
g        6
dtype: int64
```

D)

	0	1
a	0.0	0.0
c	NaN	NaN
b	1.0	1.0
e	NaN	NaN

Concatenation through inner join

Intersect them by passing join='inner’

```
s1 = pd.Series([0, 1], index=['a', 'b’])
s4 = pd.Series([0, 1, 5, 6], index=['a','b','f', 'g’])
pd.concat([s1, s4], axis=1, join='inner’)
```

	0	1
a	0	0
b	1	1

Specify the axes to be used on the other axes with

```
join_axes:
pd.concat([s1, s4], axis=1,
    join_axes=[['a', 'c', 'b', 'e']])
```

	0	1
a	0.0	0.0
c	NaN	NaN
b	1.0	1.0
e	NaN	NaN

Concatenation issues

Results are not identifiable

Solution: use hierarchical indexing

```
result = pd.concat([s1, s1, s3], keys=['one',
'two', 'three'])
```

one	a	0
	b	1
two	a	0
	b	1
three	f	5
	g	6

`Result.unstack:`

	a	b	f	g
one	0.0	1.0	NaN	NaN
two	0.0	1.0	NaN	NaN
three	NaN	NaN	5.0	6.0

Concatenation issues cont.

Along axis =1 the keys become column headers

```
result = pd.concat([s1, s1, s3], keys=['one',
'two', 'three’], axis = 1)
```

	one	two	three
a	0.0	0.0	NaN
b	1.0	1.0	NaN
f	NaN	NaN	5.0
g	NaN	NaN	6.0

`Result.unstack:`

one	a	0.0
	b	1.0
	f	NaN
	g	NaN
two	a	0.0
	b	1.0
	f	NaN
	g	NaN
three	a	NaN
	b	NaN
	f	5.0
	g	6.0

Concatenate DataFrames

Same logic applies to DataFrame concatenation

```
pd.concat([df1, df2], axis=1, keys=['level1', 'level2'], sort =True, names=['upper','lower'])
```

df1

	one	two
a	0	1
b	2	3
c	4	5

df2

	three	four
a	5	6
c	7	8

Create hierarchical index using `keys` attribute

	level1		level2	
	one	two	three	four
a	0	1	5.0	6.0
b	2	3	NaN	NaN
c	4	5	7.0	8.0

Name the created axis levels:

upper	level1		level2	
lower	one	two	three	four
a	0	1	5.0	6.0
b	2	3	NaN	NaN
c	4	5	7.0	8.0

Important NOTE

Pandas concatenation *preserves indices*, even if the result will have duplicate indices! Use verify_integrity attribute to catch errors

```
try:
    pd.concat([df1, df2],
      join_axes=[df1.columns],
      verify_integrity=True,sort = True)
except ValueError as e:
    print("ValueError:", e)
```

	one	two
a	0.0	1.0
b	2.0	3.0
c	4.0	5.0
a	NaN	NaN
c	NaN	NaN

df1

df2

```
ValueError: Indexes have overlapping values: Index(['a', 'c'], dtype='object')
```

Important NOTE

If indices are not important, turn the flag ignore_index on to create new indices for the concatenated DataFrame

```
pd.concat([df1, df2] ,join_axes=[df1.columns], ignore_index=True)
```

df1

df2

	one	two
a	0.0	1.0
b	2.0	3.0
c	4.0	5.0
a	NaN	NaN
c	NaN	NaN

	one	two
0	0.0	1.0
1	2.0	3.0
2	4.0	5.0
3	NaN	NaN
4	NaN	NaN

Alternation and optimization

Use `append` instead of `concat()`: `df1.append(df2)`

`Append` method in Pandas

- Does not modify the original object
- Is not an efficient method

Optimization:

Build a list of DataFrames

Pass them all at once to the `concat()` function.

Combining data with overlap

You can combine the `NaN` values of a DataFrame with values from another DataFrame

```
df1.combine_first(df2)
```

df1

	a	b	c
0	1.0	NaN	2
1	NaN	2.0	6
2	5.0	NaN	10
3	NaN	6.0	14

df2

	a	b
0	5.0	NaN
1	4.0	3.0
2	NaN	4.0
3	3.0	6.0
4	7.0	8.0

df1.combine_first(df2)

	a	b	c
0	1.0	NaN	2.0
1	4.0	2.0	6.0
2	5.0	4.0	10.0
3	3.0	6.0	14.0
4	7.0	8.0	NaN

Similar operation in numpy is where: np.where(pd.isnull(a), b, a)

It uses b values for a entries that are null

Data Grouping and Aggregation

UBCO Master of Data Science – DATA 542

Introduction

Gain more insights to your data

Summarize data

Statistics of data

Pandas objects aggregation method

Aggregation methods for Pandas DataFrame and Series objects:

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

Aggregations

Aggregation functions:

`sum()`, `min()`, `max()`, `mean()`

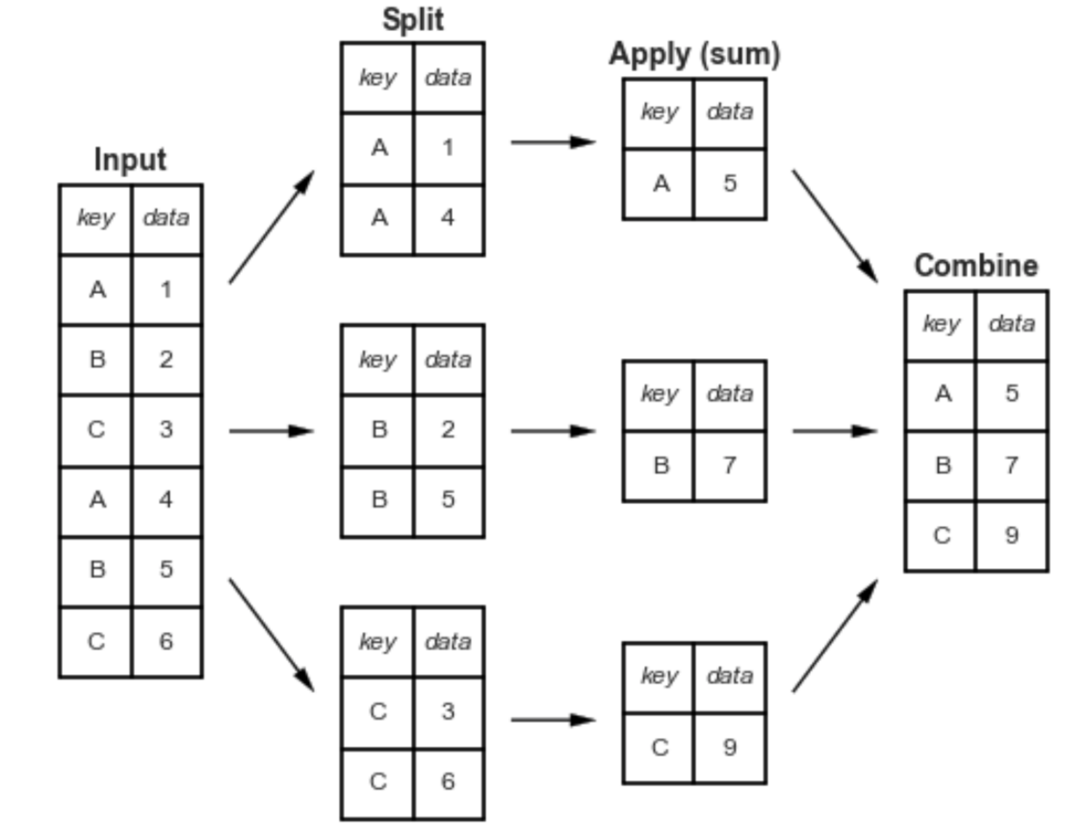
In `numpy` arrays, the aggregations output a single value

In `Pandas Series`, the aggregations output a single value

In `DataFrames`, the aggregations are operated in each column separately

You can set the `axis=1` or `axis='columns'` to operate the aggregation functions on each row

Deeper analytics using Groupby



Picture soruce: https://jakevdp.github.io/PythonDataScienceHandbook/03.08-aggregation-and-grouping.html

Lazy evaluation

`groupby(column_name)` is the basic method:

`df.groupby('key')`

	key	data
0	A	0
1	B	1
2	C	2
3	A	3
4	B	4
5	C	5

```
<pandas.core.groupby.groupby.DataFrameGroupBy object at 0x11e7d9b38>
```

Lazy evaluation: Nothing is done until an actual apply operation

`df.groupby('key').sum()`

	data
key	
A	3
B	5
C	7

Try it yourself

Planet dataset from seaborn

It gives information on planets that astronomers have discovered around other stars (known as *extrasolar planets* or *exoplanets* for short).

Load the data:

```
import seaborn as sns
planets = sns.load_dataset('planets')
planets.shape
```

Take a moment to review the data in Pandas

Explore aggregation functions on the `planets` DataFrame or one column of the DataFrame

Quickly gain insights to a dataset

Use describe method on a DataFrame to get common aggregate functions: `planets.dropna().describe()`

	number	orbital_period	mass	distance	year
count	498.00000	498.000000	498.000000	498.000000	498.000000
mean	1.73494	835.778671	2.509320	52.068213	2007.377510
std	1.17572	1469.128259	3.636274	46.596041	4.167284
min	1.00000	1.328300	0.003600	1.350000	1989.000000
25%	1.00000	38.272250	0.212500	24.497500	2005.000000
50%	1.00000	357.000000	1.245000	39.940000	2009.000000
75%	2.00000	999.600000	2.867500	59.332500	2011.000000
max	6.00000	17337.500000	25.000000	354.000000	2014.000000

Discuss the results. What do you understand from this?

Groupby object

It is an abstraction of your data

Index a specific column with `groupby`: Referencing to a grouped Series by its column name

```
planets.groupby('method')['orbital_period'].median() #.count(), .sum(), etc…
```

```
method
Astrometry                         2
Eclipse Timing Variations          9
Imaging                           12
Microlensing                       7
Orbital Brightness Modulation      3
Pulsar Timing                      5
Pulsation Timing Variations        1
Radial Velocity                  553
Transit                          397
Transit Timing Variations          3
Name: orbital_period, dtype: int64
```

Try It

Try an example from the *Python Data Science Handbook* *by Jake VanderPlas*

Go to the "Example: US States Data" section of the book: https://jakevdp.github.io/PythonDataScienceHandbook/03.07-merge-and-join.html

Download the datasets and work through the example given in the book.


UBC
THE UNIVERSITY OF BRITISH COLUMBIA