Aggregating Data

1405 Instructor: Ruiqing (Sam) Cao

Aggregate Data to a Coarser Level

- First, choose the **primary key** for the *aggregated data*. This key will uniquely identify each group in the aggregated dataset

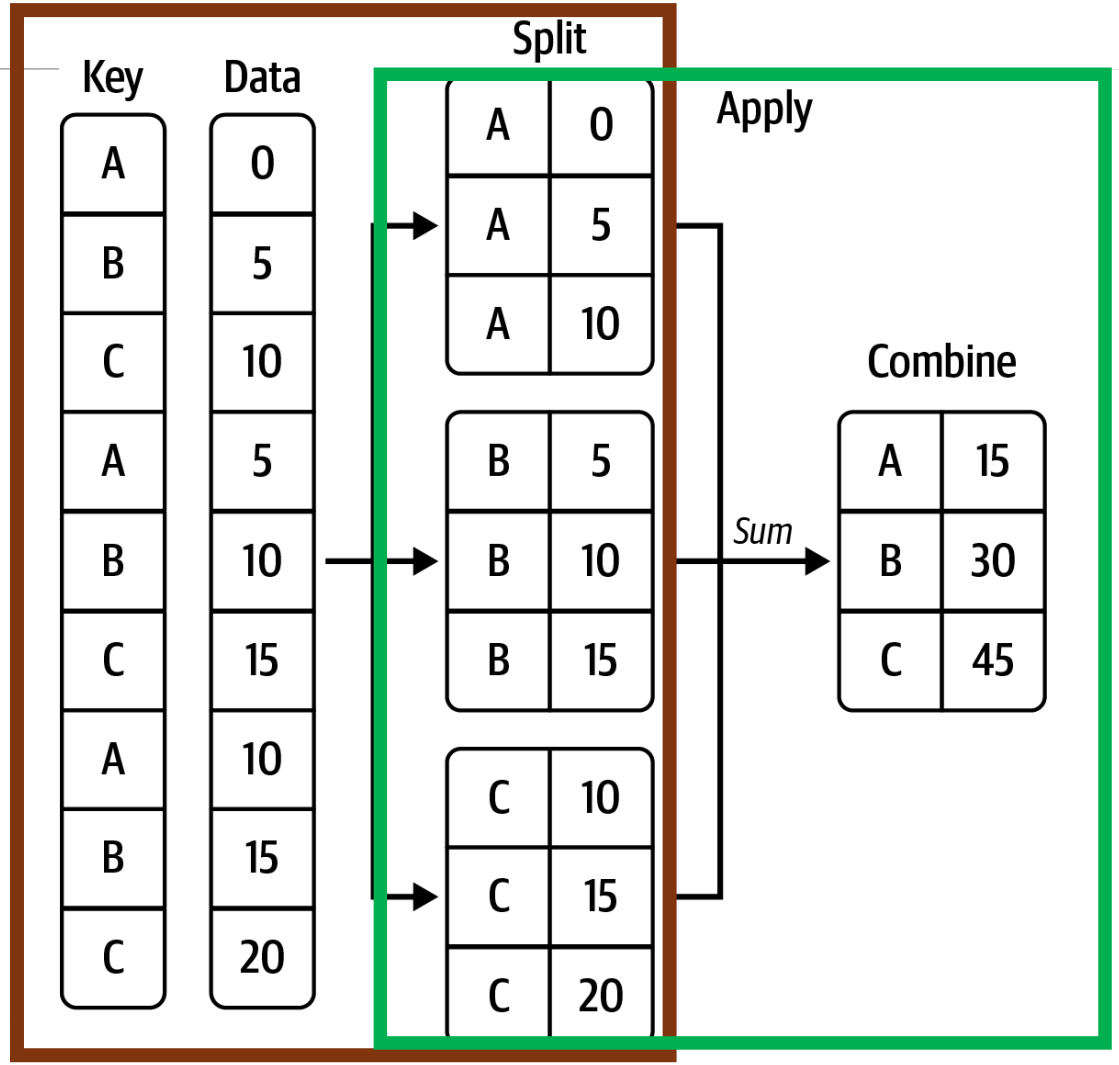
Data aggregation involves two operations

- **Group By**: Group the data based on the new primary key. Each group will consist of observations that share the same primary key
- **Aggregate**: Apply an aggregation function (e.g., `sum()`, `mean()`, or `count()`) to each group. The result is a single row per group with the aggregated values

→ Aggregated dataset where each group corresponds to a unique primary key and its aggregated values

Group By and Aggregate

Group By



Aggregate

McKinney, Figure 10.1: Illustration of a group aggregation

Group By and Aggregate

	`DataFrame.groupby(keys).agg_func(data)`
Arguments	`keys`: a list of one or more column names indicating the primary keys of the aggregated data `data`: sometimes N/A which means aggregation is applied to all columns; otherwise, one or more column names on which `agg_func()` is applied
agg_func	The aggregation function e.g., `mean()`, `sum()`, `size()`, `count()`, `min()`, `max()`, `std()`, `var()`, `quantile()`
Returns	a DataFrame object indexed by `keys` and with exactly one row per value of the `keys`

Commonly Used Aggregation Functions

- `count()`: *cannot take in any argument*, returns the number of non-missing values for ALL columns (except for the key) in each group (relatedly `nunique()` counts unique non-missing values)

- `size()`: *cannot take in any argument*, returns the total number of observations in each group (regardless of missing values)

- `mean()`, `sum()`, `min()`, `max()`, `std()`, `var()`: *takes no argument*, returns aggregated values in each group for all numeric column(s)

Group By and Aggregate: an Example

Count the number of users joining Venmo for each unique *(year, month, is_group)*

Solve the task using:

- `size()`
- `count()`
- `sum()`

as the aggregation function

	first_name	is_group	id	date_joined	joined2018	year	month	day
0	Sion	False	2082497001160704615	2016-11-13T07:09:48	False	2016	11	13
1	Kari	False	2538731244355584742	2018-08-04T18:45:48	True	2018	08	04
2	Jessie	False	1921315569139712500	2016-04-04T21:51:04	False	2016	04	04
3	Dion	False	2019594260709376161	2016-08-18T12:13:17	False	2016	08	18
4	Alec	False	2080895330680832565	2016-11-11T02:07:34	False	2016	11	11
...	...	...	...	...	...	...	...	...
19995	Matthew	False	1774670361657344464	2015-09-15T13:53:13	False	2015	09	15
19996	Matt	False	1080572944318464679	2013-01-30T21:45:33	False	2013	01	30
19997	Jerson	False	1874983190003712396	2016-01-31T23:36:54	False	2016	01	31
19998	Xinrong	False	2046581142454272653	2016-09-24T17:51:24	False	2016	09	24
19999	Paul	False	1333574510837760138	2014-01-14T23:34:30	False	2014	01	14

19992 rows × 8 columns

Group By and Aggregate: an Example

The row indexes of the aggregated DataFrame is its **primary key**

Use `size()`:

```
pd.DataFrame(data.groupby(['year','month',
'is_group']).size(),columns=['cnt'])
```

Use `count()`:

```
data.groupby(['year','month','is_group'])
[['id']].count().rename(columns={'id':'cnt'})
```

Use `sum()`:

```
data['cnt'] = 1
data.groupby(['year','month','is_group'])[[
'cnt']].sum()
```

			cnt
year	month	is_group	
2011	02	False	1
	11	False	1
2012	01	False	1
	02	False	2
	04	False	1
...	...	...	...
2018	05	False	393
		True	1
	06	False	430
	07	False	558
	08	False	1

96 rows × 1 columns

Multiple Aggregation Functions at Once

If you want to calculate the sum of some variables and the average of others, using only one aggregation function at a time and then concatenating the results can be very cumbersome

You can use `DataFrame.groupby().agg(dict)` to perform different types of aggregation at once

- The argument `dict` is **a dictionary mapping each column to one or a list of functions** (literals such as `'sum'`) to apply to that column

You can **chain it with `.rename()`** or **`.set_axis()`** to fix the column names

Flatten Index After Data Aggregation

	`DataFrame.reset_index()`
Arguments	N/A
Returns	a new DataFrame with default index, and the index of the original DataFrame become separate columns

- The method can also be performed *inplace*, for example:

```
data.reset_index(inplace=True)
```

Modifies `data` directly by resetting its index to default and turning the original index to separate columns

Flatten Index After Data Aggregation

			count
year	month	is_group	
2011	02	False	1
	11	False	1
2012	01	False	1
	02	False	2
	04	False	1
...	...	...	...
2018	05	False	393
		True	1
	06	False	430
	07	False	558
	08	False	1

96 rows × 1 columns

Old index

```
data.reset_index(inplace=True)
```

	year	month	is_group	count
0	2011	02	False	1
1	2011	11	False	1
2	2012	01	False	1
3	2012	02	False	2
4	2012	04	False	1
...	...	...	...	...
91	2018	05	False	393
92	2018	05	True	1
93	2018	06	False	430
94	2018	07	False	558
95	2018	08	False	1

New index

96 rows × 4 columns

Exercise: Data Aggregation

(Previous Exercise) Load assignment1_venmo_dataset_jul2018.csv into a Pandas DataFrame. Create a **transactions table** that satisfies **1NF** and includes only columns you consider important

Aggregate the transactions table at the level of **(year, month, day)** to include the following metrics:

1. Total number of transactions each day
2. Number of “Charge” transactions
3. Number of “Pay” transactions
4. Number of unique users initiating a “Pay” transaction
5. Number of unique users initiating a “Charge” transaction
6. Average number of transactions initiated per user