C: Grouping and aggregating data with Pandas

Computing statistics for groups within a dataset

Done in two distinct steps:

1. Grouping
   Collect records into groups

   - Purely organizational

2. Aggregating
   Compute group values from member data

   - Applies a given calculation to each group
   - Collapses the data: one value per group rather than per member

Implementation in Pandas:

1. **Grouping:** build grouped data using **.groupby()** method.

2. **Aggregating:** apply aggregation functions to the grouped data.

Example:

Step 0: Initial data

raw, index = id:

id	type	inc	age
1	A	50	32
2	B	80	40
3	B	30	20
4	A	70	27
5	B	88	50

Step 1: grouping

```
grouped = raw.groupby('type')
```

Returns a GroupBy object that organizes rows into groups:

type A:

id	type	inc	age
1	A	50	32
4	A	70	27

type B:

id	type	inc	age
2	B	80	40
3	B	30	20
5	B	88	50

Step 2: aggregating

Applying one or more functions to the GroupBy object

Examples: applying functions to **individual variables**

```
mean_inc = grouped['inc'].mean()
```

type	inc
A	60
B	66

- **Index** will always be the **grouping variable(s)** from .groupby()

```
med_age = grouped['age'].median()
```

type	age
A	29.5
B	40

What aggregation functions can be applied to a GroupBy object?

1. **Series methods** that **return a scalar**; some common examples:

Method	Description
.sum()	
.quantile(0.25)	Value at the 25th percentile
.count()	Count of non-missing values in a group
.mean() and .median()	
.min() and .max()	
.std() and .var()	

- Return one record **per group**
- Analogous to collapse in Stata

2. **Descriptive methods**; some common examples:

.size()	Number of items in the group
.describe()	Descriptive statistics

- Return one record **per group**

Aside: .size() vs .count():

.size()	total number of records
.count()	records with non-missing data for a given variable

3. Methods that **return subsets of each group**; handy examples:

.head(N)	First N rows of each group
.tail(N)	Last N rows of each group
.nlargest(N)	N rows with the largest value of given column
…	

- Return **several records** per group

4. Methods that produce **information** about the **ungrouped data**; examples:

.ngroup()	Each original row's group number
.cumcount()	Original row's sequence number within its group
.cumsum()	Cumulative sum within each group

.rank() Rank of each row within its group

…

Return **one row** for **each row** in the **ungrouped** data
Can be saved back into the original DataFrame

5. Many, many others, especially via the **.agg() method:**

.agg() is a Swiss Army Knife:

- Can apply a wide range of functions (including user-written)
- Can apply multiple functions at once
- Can apply different functions to different columns

Aggregation examples:

- Applying **one** function to **all variables** in a DataFrame:

```
means = grouped.mean()
```

Function applied to every column:

type	inc	age
A	60.0	29.500000
B	66.0	36.666667

- Applying **multiple functions** to one variable:

```
details = grouped['inc'].agg( ['min', 'max'] )
```

type	min	max
A	50	70
B	30	88

Note: the argument is a list and the function names can be strings

- Applying **different functions** to **different variables**:

```
details = grouped.agg( {'inc':'max', 'age':'mean'} )
```

type	inc	age
A	70	29.500000
B	88	36.666667

Note: the argument is a dictionary with columns as keys

- Applying a **descriptive function** to one variable:

```
details = grouped['inc'].describe()
```

Generate multiple statistics per variable:

type	count	mean	std	min	25%	50%	75%	max
A	2.0	60.0	14.142136	50.0	55.0	60.0	65.0	70.0
B	3.0	66.0	31.432467	30.0	55.0	80.0	84.0	88.0

- Many other combinations are possible

Continue in g11 demo.py