Pandas

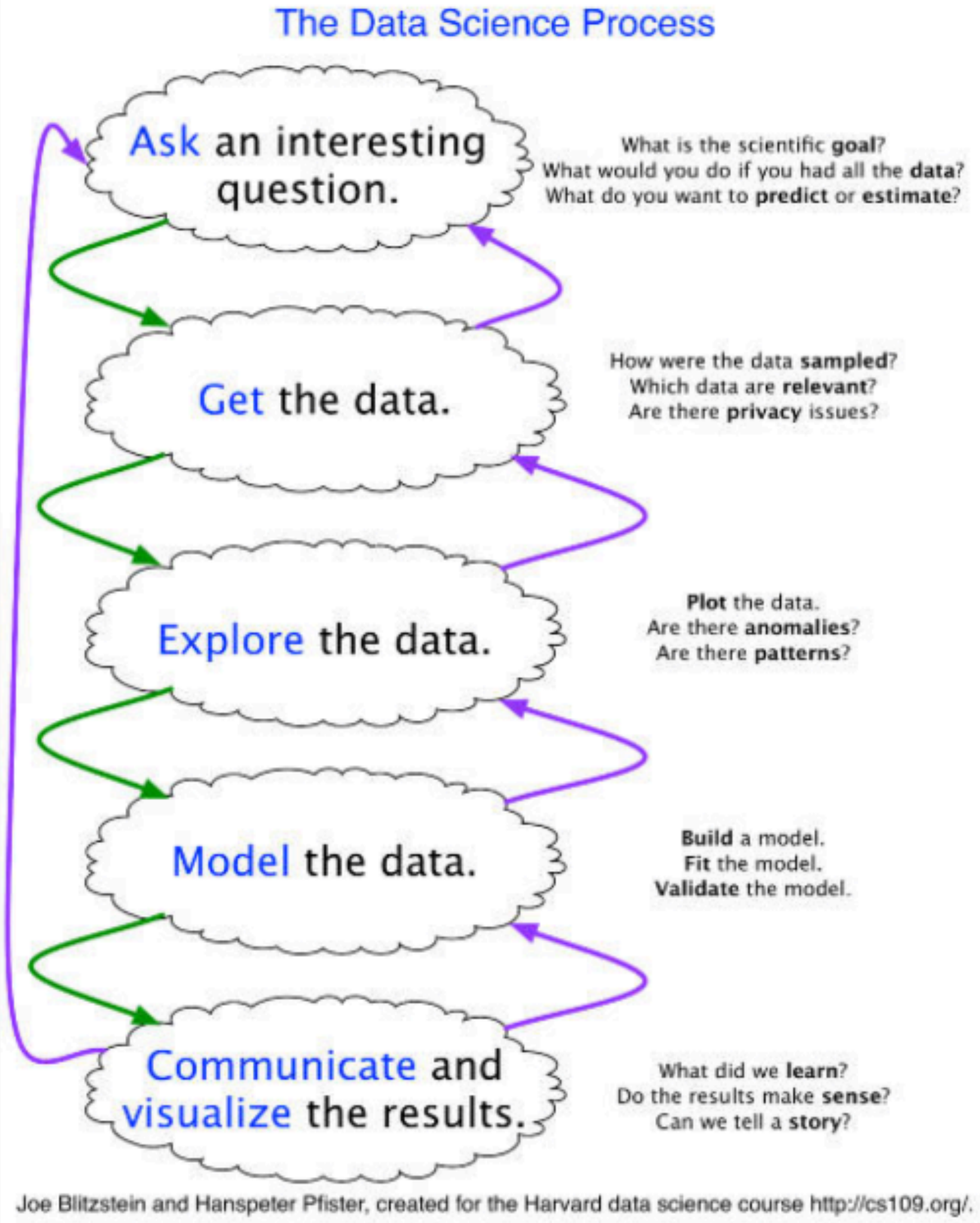
CS3300 Data Science

RJ Nowling

Readings

- Chapters 2 and 3 of the *Python Data Science Handbook*

Data Science Process



The Data Science Process

Ask an interesting question.

What is the scientific **goal**?
What would you do if you had all the **data**?
What do you want to **predict** or **estimate**?

Get the data.

How were the data **sampled**?
Which data are **relevant**?
Are there **privacy** issues?

Explore the data.

Plot the data.
Are there **anomalies**?
Are there **patterns**?

Model the data.

Build a model.
Fit the model.
Validate the model.

Communicate and visualize the results.

What did we **learn**?
Do the results make **sense**?
Can we tell a **story**?

Joe Blitzstein and Hanspeter Pfister, created for the Harvard data science course http://cs109.org/.

What is Pandas?

- Library for manipulating tables of data
- Primarily used for cleaning and restructuring data in preparation for plotting or modeling
- 3 primary data structures
    - Series – 1D, columns of data
    - DataFrames – 2D, tables of data
    - Panel – 3D, cube of data (rarely used, deprecated and going to be removed)
- Columnar
    - Most operations are designed to operate on columns of data, not individual elements or rows

Pandas vs Numpy

Numpy

- Any dimension
- Indexing by position (e.g., row or column)
- Usually a single type (e.g., int, float)

Pandas

- Limited to 1 (Series) or 2 (DataFrame) dimensions
- Indexing primarily by column names
- Each column has a its own type

Pandas and SQL

- A DataFrame is similar to a table in a SQL database
- SQL also operates on columns
- Many Pandas operations have analogs in SQL:
  - Head – select * from <table> limit 10;
  - Selecting columns – select column1, column2, … from <table>;
  - Filtering rows – where column1 > 5;
  - Grouping rows – select max(…), column2 … group by column2;
  - Joining tables – select … from table1 join table2 on table1.column1 = table2.column1;
- Pandas DataFrames have an index – this is normally the implicit numerical index

Caveats

- Pandas offers multiple ways to do things. Some ways are newer and have learned from the mistakes of the old ways. This can be confusing and frustrating
- The pandas documentation is complex and not well organized
- It can be difficult to predict when a copy is made versus a view is created – this makes optimization challenging

Creating DataFrames

- Read from a CSV file:

```
df = pd.read_csv(filename)
```

- From existing lists, Numpy, arrays, or series:

```
df = pd.DataFrame( { "column1" : [0.0, 1.0,
2.0],
                     "column2" : np_array,
                     "column3" : series } )
```

Investigating DataFrames

- `df.head()`
- `df.dtypes`
- `df.shape`

Indexing / Selecting Columns

- Pandas has multiple ways to index. The slice operator works on columns:

  `df["column name"]` – get a single column

  `df[["column1", "column2", "column3"]]` – get multiple columns

Indexing

- You can index by position (numerical index). This follows the Numpy pattern of row, then column:

  `df.iloc[5]` – get a single row

  `df.iloc[:,  :3]` – get first 3 columns and all rows of the df

  `df.iloc[1:100]` – get the second to 100th rows

Creating a New Column

- The simplest way to create a new column:

  ```
  df["new column name"] = <list, 1D numpy array,
  Series>
  ```

- The assign method is useful since its returns a new DataFrame and can be used with method chaining:

  ```
  new_df = df.assign(<new_column_name> = <list,
  1D numpy array, Series>)
  ```

Modifying a Column

- Convert data types – may need to specify function for parsing / conversion
- Cleaning data
- Extracting fields from complex types
    - e.g., hour, month, etc. from date times

Modifying a Column

1. Get the Series for the column of interest

   ```
   column = df["column name"]
   ```

2. Use the map() method to apply a function to each element in the Series an return a new Series

   ```
   converted = column.map(lambda s: s + 1)
   ```

3. Then update the df, either by adding a new column or overwriting the original column

   ```
   df["column name"] = converted
   ```

Dropping a Column

- I prefer to use the drop() method because it returns a DataFrame object so it works with chaining:

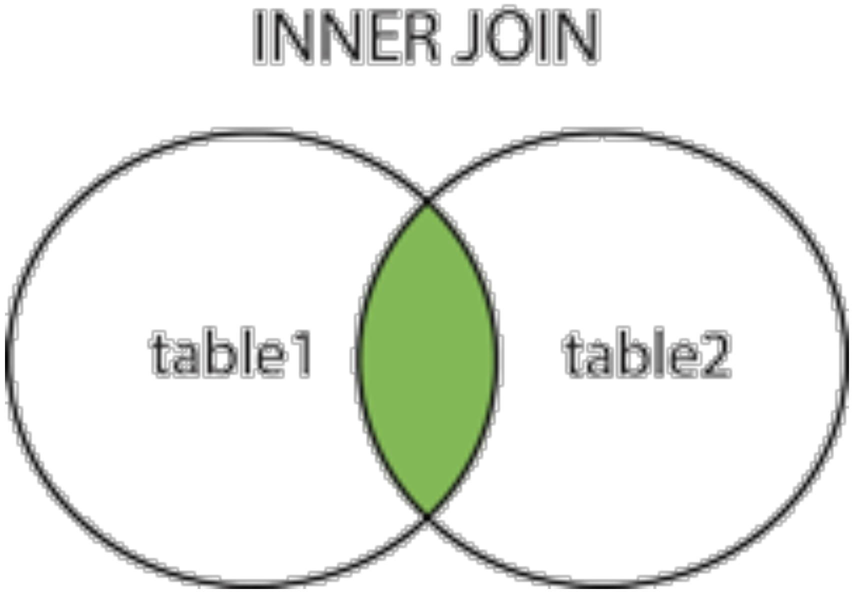
```
new_df = df.drop(columns=["column name"])
```

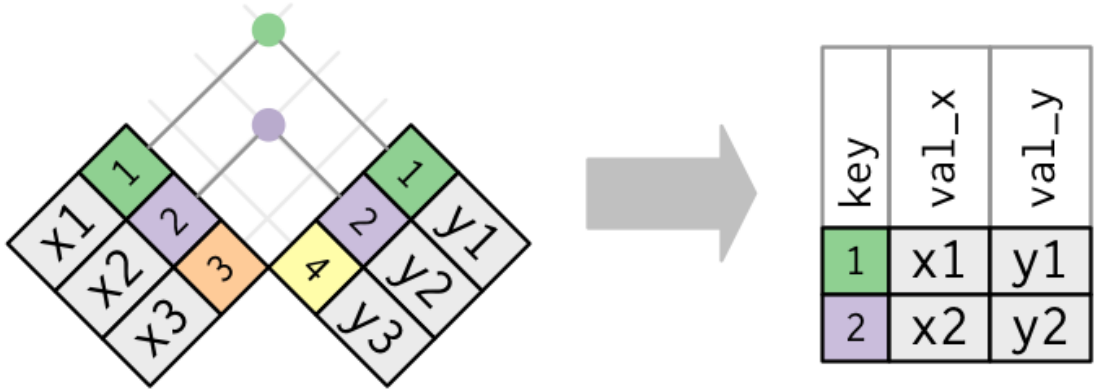
- You might see this, too:

```
del df["column_name"]
```

I like this less because it updates the DataFrame in place.

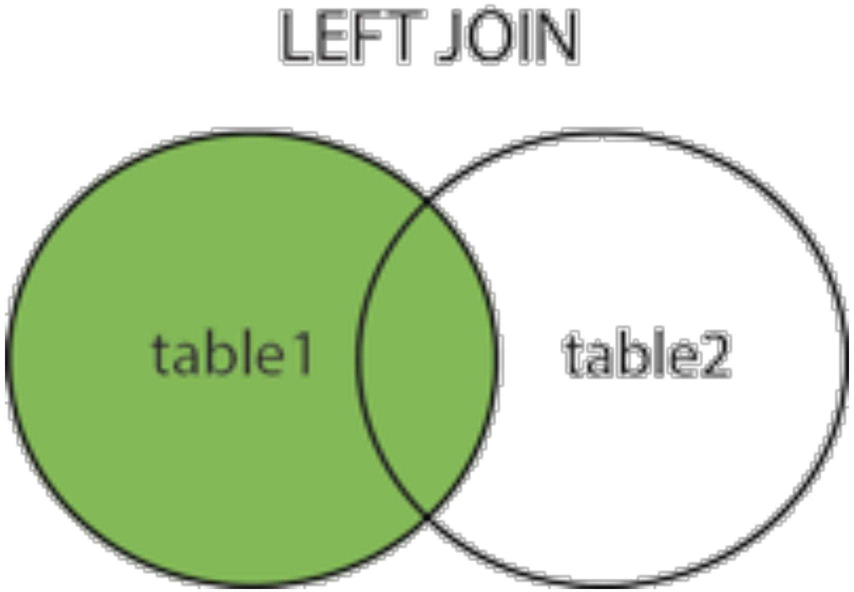
Inner Joins

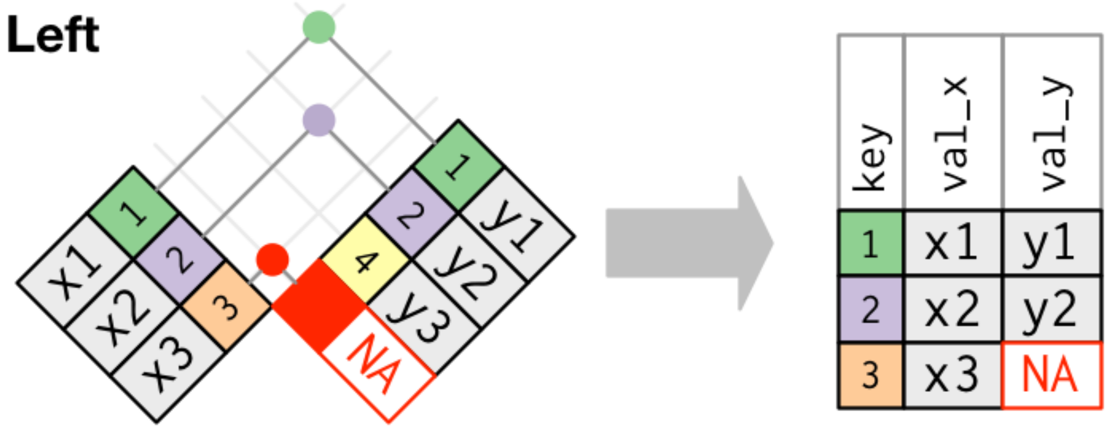




key	val_x	val_y
1	x1	y1
2	x2	y2

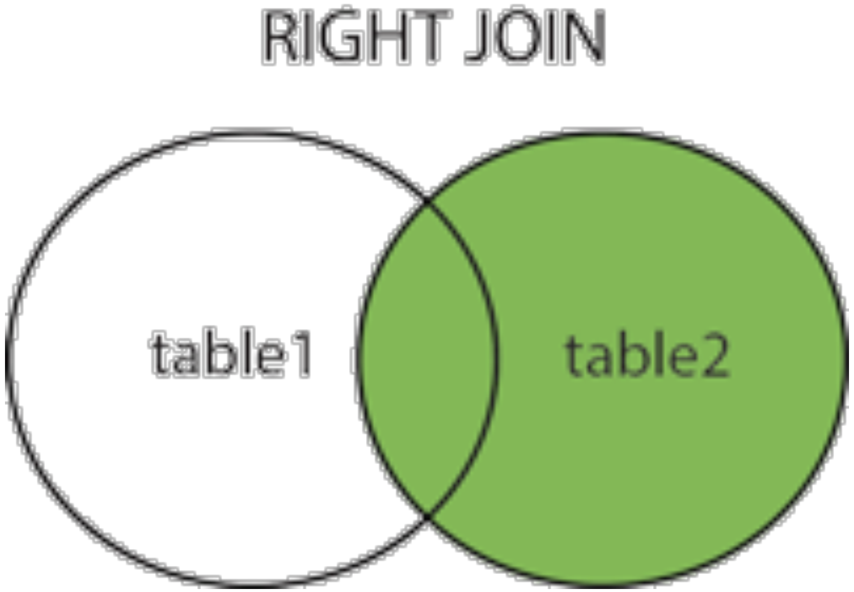
Left Outer Join

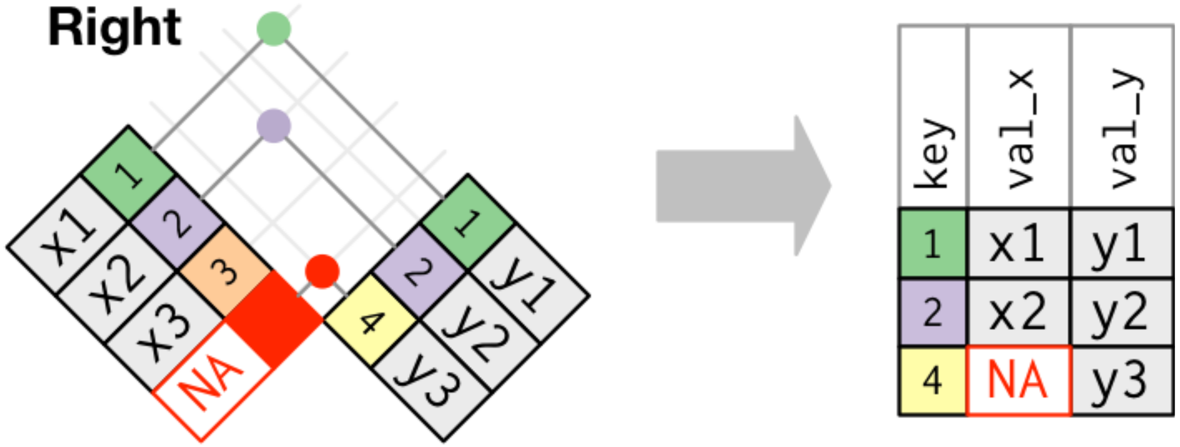




key	val_x	val_y
1	x1	y1
2	x2	y2
3	x3	NA

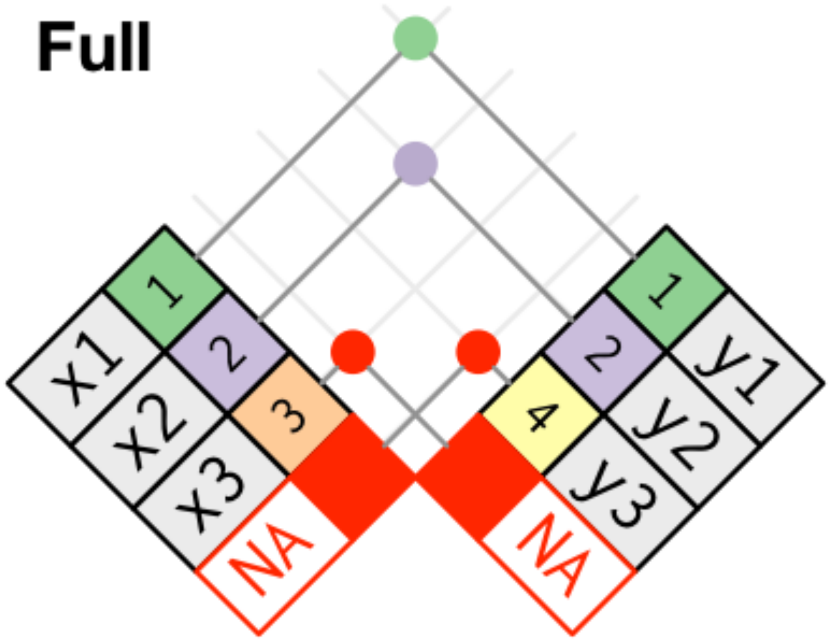
Right Outer Join

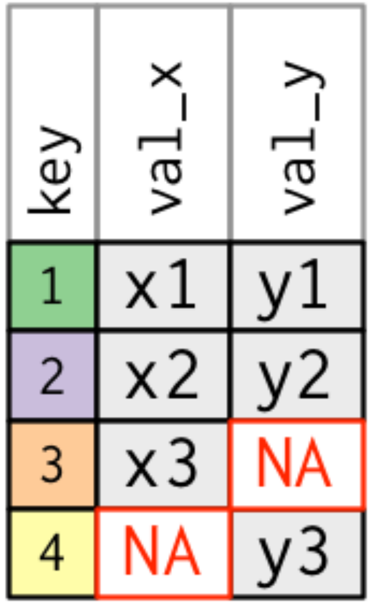




key	val1_x	val1_y
1	x1	y1
2	x2	y2
4	NA	y3

Full Outer Join





key	val_x	val_y
1	x1	y1
2	x2	y2
3	x3	NA
4	NA	y3

Pandas Join

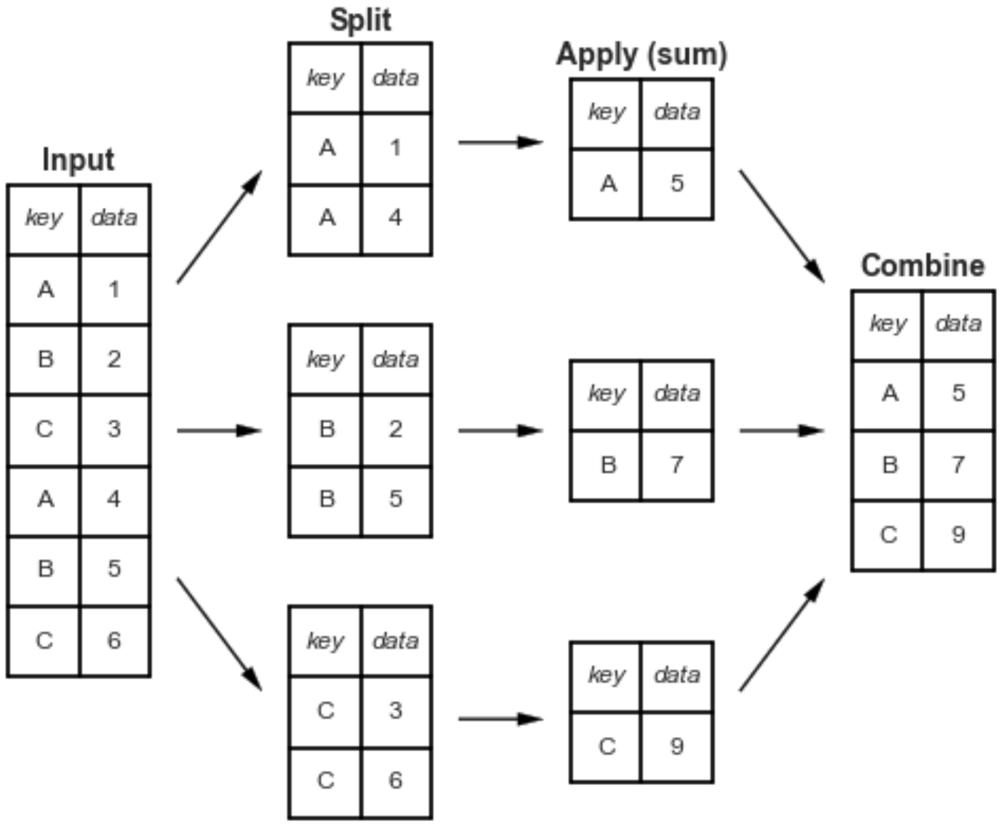
```
joined_df = df1.merge(df2,
                      on = "column",
                      how = "inner")

joined_df = df1.merge(df2,
                      left_on = "column1",
                      right_on = "column2",
                      how = "outer")
```

Group By and Aggregations

- Group by groups columns by a set of keys and aggregates the values in the remaining columns
- Two types of columns participate in group by:
  - Columns that are they keys – the keys have to have the exact same values for rows to be matched up
  - Aggregated columns – we apply count, min, max, sum, or a similar function to reduce multiple values to a single value

Group By and Aggregations



Input

key	data
A	1
B	2
C	3
A	4
B	5
C	6

Split

key	data
A	1
A	4

key	data
B	2
B	5

key	data
C	3
C	6

Apply (sum)

key	data
A	5

key	data
B	7

key	data
C	9

Combine

key	data
A	5
B	7
C	9

Group By and Aggregations

My preferred recipe:

1. Select the columns you want to use:

```
df_to_group = df[["column1", "column2", "column3"]]
```

2. Perform the group by:

```
grouped_df = df_to_group.groupby(by = ["column1",
                                       "column2"],
                                 as_index = False)
                                .min()
```