Introduction to Pandas

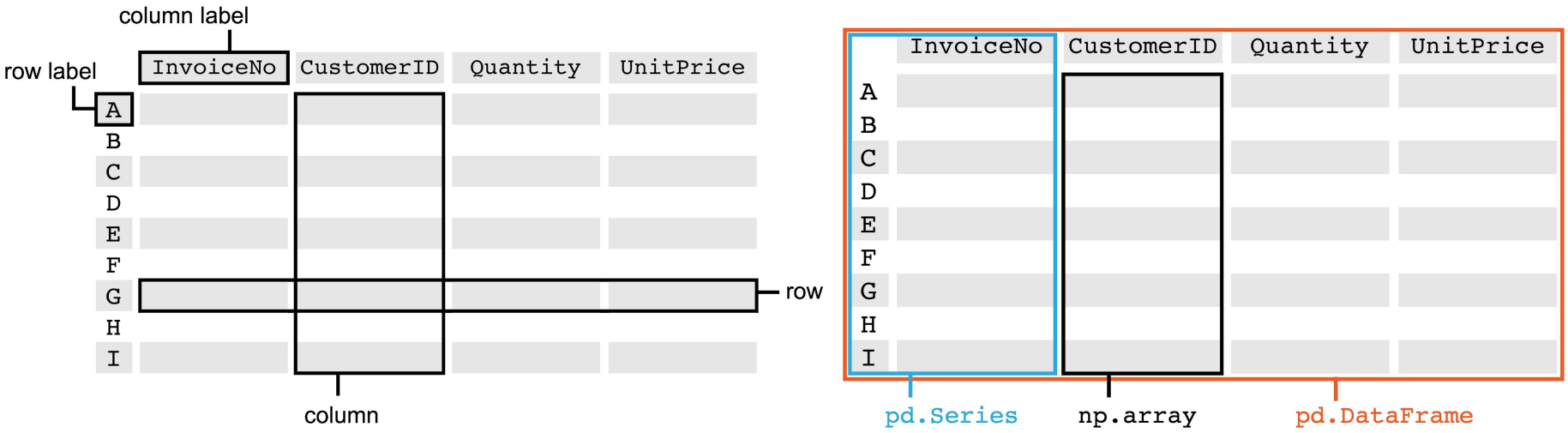
What is Pandas?

- Data **structures** and data **analysis tools**:
  - The ‘excel of python’
- Base data objects are numpy arrays (fast)
- Note: **huge** userbase - your question is on **StackOverflow**!
- Pandas **documentation** is superb
  - https://pandas.pydata.org/pandas-docs/stable/10min.html

What's a DataFrame?

- The main **datatype** in pandas
- A two-dimensional, size-mutable **table** with labeled axes (rows and columns)
- Implemented with **numpy** arrays
- It has 3 principal components: **data**, **rows** and **columns**

What's a DataFrame?



CAMBRIDGE SPARK

What's a DataFrame?

```
$  df.values
    >  array([['alice', 28],
              ['bob', 25]], dtype=object)
$  df.dtypes
$  df.columns
$  df.index
$  df.index = ['first', 'second']
$  df
    >            name  age
       first    alice   28
       second     bob   25
```

Reading data

- Pandas provides useful methods to read data from different file formats:
    - **read_csv**, **read_json**, **read_excel**
- We can specify the columns names (if not in the data already) and other read parameters (eg. separator for csv)
- Columns types can be **explicitly** set or **inferred** from data
- All these read methods return a **DataFrame**

Read from CSV

```
$  file_path = './data/customers.csv'
$  df = pd.read_csv(file_path, sep=';', header=['Name', 'Age'])


$  df.head()
```

CAMBRIDGE SPARK

What's a Series?

- Represents one column or row of a **DataFrame**
- Implemented as a numpy array with **labels** and an **index**
- All keys inside it are mapped to fields and can be accessed both ways
  - *col['name']*
  - *col.name*

Accessing elements 1/2

- **indexing**: Access columns based on their labels - *df[column]*
- **.loc**: Access elements based on row AND column labels, can return:
  - single element
  - a row/column (as a **Series**)
  - a new table (as a **DataFrame**) -
  - *Syntax: df.loc[row, column]*
- **.iloc**: Works the same way as **.loc** but uses integer indexes instead of labels
  - Syntax: *df.iloc[1, 2]*

CAMBRIDGE SPARK

Accessing elements 2/2

- **slicing**: Works the same way as with lists, we can get multiple columns or rows with this method
  - *df.loc[:, column]*
  - *df.iloc[2:4, 3:6]*
- **query:** Takes a boolean expression, returns rows that evaluate to True

Accessing elements

```
$  df['name']  # or df.name
$  df.loc['second', 'age']
$  df.iloc[1, 1]
$  df.query('age > 25')
  >          name  age
     first  alice   28
```

CAMBRIDGE SPARK

Describing data

The **.describe** method returns statistics about each numerical column of a DataFrame

```
$ df = pd.DataFrame(np.random.randn(100, 4))
$ df.describe()
  >              0       1       2       3
     count    100     100     100     100
     mean   -0.09    0.04   -0.16    0.01
     std     1.05   1.071   1.005   1.046
     min    -2.65   -2.34   -2.31   -2.16
     25%    -0.96   -0.68   -0.77   -0.75
     50%    -0.15   -0.06   -0.13   -0.08
     75%     0.65    0.69    0.54    0.68
     max     2.59    2.79    2.69    2.24
```

CAMBRIDGE SPARK

Transforming data

- **.merge**: similar to joining 2 SQL tables, useful if 2 frames have data about the same entity linked by some common feature (column):
  - columns to join on can be defined with the **left_on** and **right_on** parameters
  - join type can be defined with the **how** parameter (outer, inner, left, right)
- **.join** - a less flexible version of **.merge** which automatically joins, based on the indexes

CAMBRIDGE SPARK

.merge

```
$ import pandas as pd
$ ages = {'name': ['alice', 'bob'], 'age': [28, 25]}
$ homes = {'name': ['alice', 'bob'], 'home': ['London', 'Rome']}
$ df_ages = pd.DataFrame(ages)
$ df_homes = pd.DataFrame(homes)
$ df_ages.merge(df_homes, right_on='name', left_on='name')
  >      name  age home
     0  alice   28 London
     1    bob   25 Rome
```

CAMBRIDGE SPARK

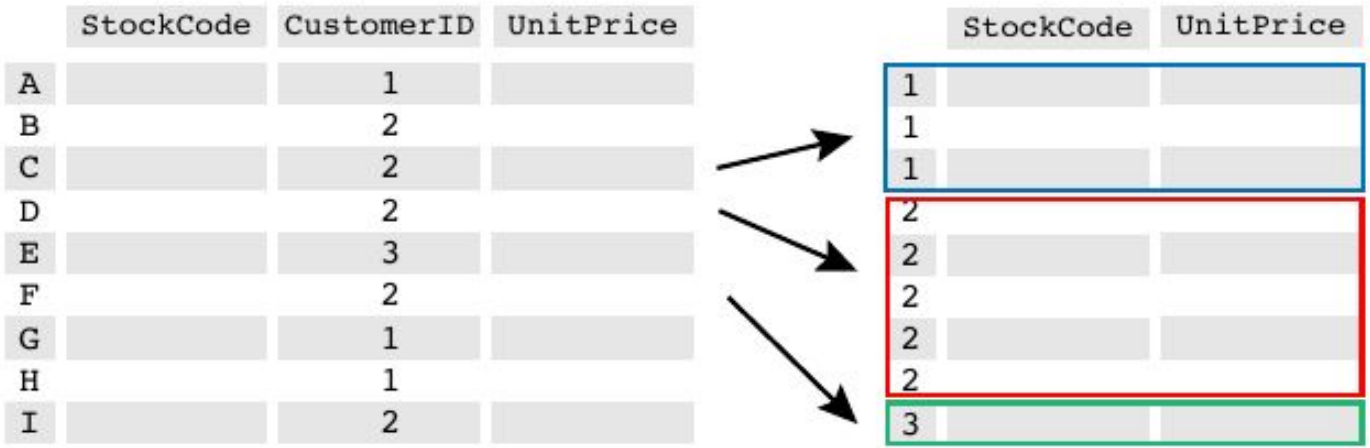
Transforming data

- **.groupby** - similar to SQL where we group according to a column
  - After grouping, the **.apply** method can be used to apply custom aggregations on the other columns
  - You can also used built-in pandas methods such as **.mean**, **.max** for the most common aggregations
  - Use **.agg** to apply multiple aggregations at once
- **.apply** either takes a lambda expression, or a numpy built-in method

CAMBRIDGE SPARK

.groupby followed by **.apply**

```
df.groupby('CustomerID')
```

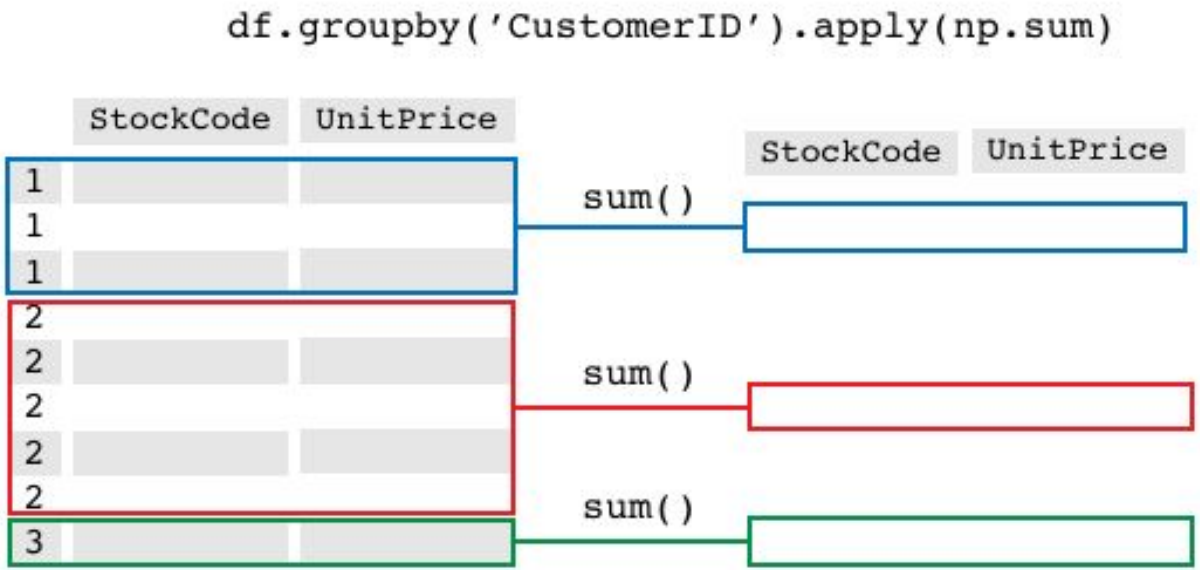


	StockCode	CustomerID	UnitPrice
A		1	
B		2	
C		2	
D		2	
E		3	
F		2	
G		1	
H		1	
I		2	

	StockCode	UnitPrice
1		
1		
1		
2		
2		
2		
2		
2		
3		

CAMBRIDGE SPARK

.groupby followed by **.apply**



CAMBRIDGE SPARK

Filtering data

Inserting a **DataFrame** column into a boolean expression returns **Series** of True/False values, this is known as a mask and can be used to index your **DataFrame**

```
$ df = {'name': ['Alice', 'Bob', 'Kevin'], 'age': [24, 16, 28]}
$ mask = df['age'] > 25
$ print(mask)
  > 0 False
    1 False
    2 True
$ df.loc[mask]
  >     name   age
    2   Kevin  28
```

Saving data

- Similarly to reading data, pandas provides us with built-in functions to save our **DataFrames** to files
- As with input, we can output to several different formats
  - **to_csv**, **to_json**
- specifying the same types of parameters as with reading (eg. filename, separator)

Hands-on session

pandas.ipynb

CAMBRIDGE SPARK