CS112
Introduction to Python Programming

Session 11: Pandas II

Shengwei Hou
Ph.D., Assistant Professor
Department of Ocean Science and Engineering
Fall 2022

SUSTech Southern University of Science and Technology

明德求是 日新自强
VIRTUE | TRUTH | ADVANCE

Contents

Introductions

- `Pandas` has two principal data structures: `Series` and `DataFrame`, which form the basis for most activities using `Pandas`.
- The two primary data structures of pandas, `Series` (one-dimensional) and `DataFrame` (two-dimensional), handle the vast majority of typical use cases
- `Series` is a one-dimensional array that can store various data types, including mixed data types
- Any list, tuple, and dictionary can be converted into `Series` objects
- The basic method to create a `Series` is to call:

```
pd.Series(data, index=index)
```

- The axis labels in a `Series` are collectively referred to as the `index`.

```
>>> import pandas as pd
>>> s = pd.Series(data, index=None)
```

Here, `s` is a `Pandas Series`, data can be a Python `dict`, a `ndarray`, or a scalar value (like 5). The passed `index` is a list of axis labels.

- Both integer and label-based indexing are supported. If the `index` is not provided, then the `index` will default to `range(n)` where `n` is the length of data.

Series

Southern University of Science and Technology

- Create `Series` from ndarrays:

```
>>> import numpy as np
>>> import pandas as pd
>>> s = pd.Series(np.random.randn(5), index=['a',
'b', 'c', 'd', 'e'])
>>> type(s)
<class 'pandas.core.series.Series'>
>>> s
a -0.367740
b 0.855453
c -0.518004
d -0.060861
e -0.277982
dtype: float64
>>> s.index
Index(['a', 'b', 'c', 'd', 'e'], dtype='object')
>>> s.values
array([-0.367740, 0.855453, -0.518004, -0.060861, -
0.277982])
>>> pd.Series(np.random.randn(5))
0 0.334947
1 -2.184006
2 -0.209440
3 -0.492398
4 -1.507088
dtype: float64
```

index

values

- You can specify axis labels for `index`, i.e., `index=['a', 'b', 'c', 'd', 'e']`.
- When data is a `ndarray`, the `index` must be the same length as data. In series `s`, by default the type of values of all the elements is `dtype: float64`.
- You can find out the index for a series using `index` attribute. The `values` attribute returns a `ndarray` containing only values, while the axis labels are removed.
- If no labels for the index is passed, one will be created having a range of index values `[0, ..., len(data) - 1]`.

- Create `Series` from Dictionaries:

```
>>> import numpy as np
>>> import pandas as pd
>>> d = {'a' : 0., 'b' : 1., 'c' : 2.}
>>> pd.Series(d)
a 0.0
b 1.0
c 2.0
dtype: float64
>>> pd.Series(d, index=['b', 'c', 'd', 'a'])
b 1.0
c 2.0
d NaN
a 0.0
dtype: float64
```

- When a series is created using dictionaries, by default the keys will be `index` labels.
- While creating a series using a dictionary, if labels are passed for the `index`, the values corresponding to the labels in the index will be pulled out. The order of index labels will be preserved.
- If a value is not associated for a label, then `NaN` is printed. `NaN` (not a number) is the standard missing data marker used in `pandas`.

Vectorized operations

- `Series` can also be passed into most `NumPy` methods
- Vectorized operations and label alignment with `Series`:

```
a = np.array(np.random.randint(1,10, size=3))
s = pd.Series(a,index=['Gene1','Gene2','Gene3'])
```

```
s
```

```
Gene1    8
Gene2    8
Gene3    1
dtype: int64
```

```
s + s
```

```
Gene1    16
Gene2    16
Gene3     2
dtype: int64
```

```
import numpy as np
np.square(s)
```

```
Gene1    64
Gene2    64
Gene3     1
dtype: int64
```

```
s + 1
```

```
Gene1    9
Gene2    9
Gene3    2
dtype: int64
```

- A key difference between `Series` and `ndarray` is that operations between `Series` automatically align the data based on labels.

```
a1 = np.array(np.random.randint(1,10, size=3))
s1 = pd.Series(a1,index=['Gene1','Gene2','Gene3'])

a2 = np.array(np.random.randint(1,10, size=3))
s2 = pd.Series(a2,index=['Gene1','Gene3','Gene2'])
```

```
s1
```

```
Gene1    8
Gene2    4
Gene3    1
dtype: int64
```

```
s2
```

```
Gene1    1
Gene3    6
Gene2    2
dtype: int64
```

```
s1 + s2
```

```
Gene1    9
Gene2    6
Gene3    7
dtype: int64
```

DataFrame

- `DataFrame` is a two-dimensional, labeled data structure with columns of potentially different types.
- `DataFrame` accepts many different kinds of input like `Dict` of one-dimensional `ndarrays`, `lists`, `dicts`, or `Series`, two-dimensional `ndarrays`, a dictionary of `Series`, or another `DataFrame`.

```
df = pd.DataFrame(data=None, index=None, columns=None)
```

- - Here, `data` can be `NumPy ndarray`, `dict`, or `DataFrame`.
- - Along with the data, you can optionally pass an `index` (row labels) and `columns` (column labels) attributes as arguments.
- - Both `index` and `columns` will default to `range(n)` where `n` is the length of data, if they are not provided.
- - When the `data` is a dictionary and `columns` are not specified, then the `DataFrame` column labels will be dictionary’s `keys`.

Create a DataFrame

- A `DataFrame` can be created from a Dictionary of Series/Dictionaries/Lists:

```
>>> import pandas as pd
>>> dict_series = {'one' : pd.Series([1., 2., 3.], index=['a', 'b', 'c']),
                   'two' : pd.Series([1., 2., 3., 4.], index=['a', 'b', 'c', 'd'])}
>>> df = pd.DataFrame(dict_series)
>>> df.shape
(4, 2)
>>> df.index
Index(['a', 'b', 'c', 'd'], dtype='object')
>>> df.columns
Index(['one', 'two'], dtype='object')
>>> list(df.columns)
['one', 'two']
```

index	one	two
a	1.0	1.0
b	2.0	2.0
c	3.0	3.0
d	NaN	4.0

- If the number of labels specified in the various series are not the same, then the resulting index will be the union of all the index labels of various series.
- Get the index labels for the DataFrame using `index` attribute. With `columns` attribute, you get all the columns of the DataFrame.

```
a = {'sample1':{'gene1':1,'gene2':2,'gene3':3},
     'sample2':{'gene1':2,'gene2':3,'gene3':4}}
pd.DataFrame(a)
```

	sample1	sample2
gene1	1	2
gene2	2	3
gene3	3	4

```
a = {'sample1':[1,2,3],'sample2':[2,3,4]}
pd.DataFrame(a,index = ['gene1','gene2','gene3'])
```

	sample1	sample2
gene1	1	2
gene2	2	3
gene3	3	4

Create a DataFrame

SUSTech Southern University of Science and Technology

- A `DataFrame` can be created using the `Pandas DataFrame` routine based on list-like objects such as `list`, `NumPy array`, or `dictionary`:

```
>>> optmat = {'mat': ['silica', 'titania', 'PMMA', 'PS'], 'index': [1.46, 2.40, 1.49, 1.59],
'density': [2.03, 4.2, 1.19, 1.05]}
>>> omdf = pd.DataFrame(optmat)
>>> omdf
       mat  index  density
0   silica   1.46     2.03
1  titania   2.40     4.20
2     PMMA   1.49     1.19
3       PS   1.59     1.05
```

```
a = [[1,2,3],[2,3,4]]
pd.DataFrame(a,index=['sample1','sample2'],
             columns=['gene1','gene2','gene3'])
```

	gene1	gene2	gene3
sample1	1	2	3
sample2	2	3	4

- The column order can be changed:

```
>>> omdf = pd.DataFrame(optmat, columns=['index', 'mat', 'density'])
>>> omdf
   index      mat  density
0   1.46   silica     2.03
1   2.40  titania     4.20
2   1.49     PMMA     1.19
3   1.59       PS     1.05
```

```
a = [(1,2,3),(2,3,4)]
df = pd.DataFrame(a)
df
```

	0	1	2
0	1	2	3
1	2	3	4

Reading & Writing

SUSTech Southern University of Science and Technology

- Reading and writing text and binary files using `pandas`:

Format Type	Data Description	Reader	Writer
text	CSV	read_csv	to_csv
text	Fixed-Width Text File	read_fwf	
text	JSON	read_json	to_json
text	HTML	read_html	to_html
text	LaTeX		Styler.to_latex
text	XML	read_xml	to_xml
text	Local clipboard	read_clipboard	to_clipboard

Format Type	Data Description	Reader	Writer
binary	MS Excel	read_excel	to_excel
binary	OpenDocument	read_excel	
binary	HDF5 Format	read_hdf	to_hdf
binary	Feather Format	read_feather	to_feather
binary	Parquet Format	read_parquet	to_parquet
binary	ORC Format	read_orc	
binary	Stata	read_stata	to_stata
binary	SAS	read_sas	
binary	SPSS	read_spss	
binary	Python Pickle Format	read_pickle	to_pickle
SQL	SQL	read_sql	to_sql
SQL	Google BigQuery	read_gbq	to_gbq

https://pandas.pydata.org/pandas-docs/stable/user_guide/io.html

Reading & Writing

```
pd.read_csv (filepath, sep=',' , header = 'infer', names = None,
index_col = None,…)
```

filepath: a path to a file
sep: Delimiter to use; ‘,’for CSV, ‘ \t’ for tab-delimited file
header: Row number(s) to use as the column names. None if file contains no header row
names: List of column names to use
index_col: Column(s) to use as the row labels of the DataFrame, either given as string name or column index.

```
df.to_csv (filepath, sep=',' , header = True, index = True, …)
```

sep : str, default ‘,’ ; ‘\t’ for tab-delimited file
header : bool or list of str, default True Write out the column names
index : bool, default **True** Write row names (index).

Viewing Data

SUSTech Southern University of Science and Technology

- Use `DataFrame.head()` and `DataFrame.tail()` to view the top and bottom rows of the frame, respectively.
- Row index and column names can be obtained via the `DataFrame.index` and `DataFrame.columns`, respectively.
- `DataFrame.describe()` shows a quick statistic summary of your data:

```
>>> import pandas as pd
>>> import numpy as np
>>> dates = pd.date_range("20130101", periods=6)
>>> df = pd.DataFrame(np.random.randn(6, 4),
index=dates, columns=list("ABCD"))
>>> df.head()
```

	A	B	C	D
2013-01-01	-0.023244	0.400826	-1.392608	0.782346
2013-01-02	-1.437795	-0.528897	0.940951	-1.840915
2013-01-03	-0.986200	1.120746	-0.974936	0.474923
2013-01-04	-0.763038	0.477575	-0.585073	-1.098130
2013-01-05	0.621200	-0.618775	1.692731	-0.037749

```
>>> df.describe()
```

	A	B	C	D
count	6.000000	6.000000	6.000000	6.000000
mean	-0.388181	0.006425	-0.213516	-0.312476
std	0.796491	0.770458	1.235727	0.987135
min	-1.437795	-0.812927	-1.392608	-1.840915
25%	-0.930409	-0.596305	-0.971742	-0.862430
50%	-0.393141	-0.064035	-0.773616	-0.096540
75%	0.189184	0.458387	0.559445	0.346755
max	0.621200	1.120746	1.692731	0.782346

Sorting Data

- `DataFrame.sort_index()` sorts by an axis
- `DataFrame.sort_values()` sorts by value

```
>>> df.sort_index(axis=1, ascending=False)
```

	D	C	B	A
2013-01-01	0.782346	-1.392608	0.400826	-0.023244
2013-01-02	-1.840915	0.940951	-0.528897	-1.437795
2013-01-03	0.474923	-0.974936	1.120746	-0.986200
2013-01-04	-1.098130	-0.585073	0.477575	-0.763038
2013-01-05	-0.037749	1.692731	-0.618775	0.621200
2013-01-06	-0.155331	-0.962159	-0.812927	0.259993

```
>>> df.sort_index(axis=0, ascending=False)
```

	A	B	C	D
2013-01-06	0.259993	-0.812927	-0.962159	-0.155331
2013-01-05	0.621200	-0.618775	1.692731	-0.037749
2013-01-04	-0.763038	0.477575	-0.585073	-1.098130
2013-01-03	-0.986200	1.120746	-0.974936	0.474923
2013-01-02	-1.437795	-0.528897	0.940951	-1.840915
2013-01-01	-0.023244	0.400826	-1.392608	0.782346

```
>>> df.sort_values(by="B")
```

	A	B	C	D
2013-01-06	0.259993	-0.812927	-0.962159	-0.155331
2013-01-05	0.621200	-0.618775	1.692731	-0.037749
2013-01-02	-1.437795	-0.528897	0.940951	-1.840915
2013-01-01	-0.023244	0.400826	-1.392608	0.782346
2013-01-04	-0.763038	0.477575	-0.585073	-1.098130
2013-01-03	-0.986200	1.120746	-0.974936	0.474923

```
>>> df.sort_values(by=["B", "C"], ascending=False)
```

	A	B	C	D
2013-01-03	-0.986200	1.120746	-0.974936	0.474923
2013-01-04	-0.763038	0.477575	-0.585073	-1.098130
2013-01-01	-0.023244	0.400826	-1.392608	0.782346
2013-01-02	-1.437795	-0.528897	0.940951	-1.840915
2013-01-05	0.621200	-0.618775	1.692731	-0.037749
2013-01-06	0.259993	-0.812927	-0.962159	-0.155331

Selection

- Selection by Label using `DataFrame.loc()` or `DataFrame.at()`.

```
>>> df.loc["20130102":"20130104", ["A", "B"]]
```

	A	B
2013-01-02	-1.437795	-0.528897
2013-01-03	-0.986200	1.120746
2013-01-04	-0.763038	0.477575

```
>>> df.at[dates[0], "A"]
```

-0.02324367422935155

By labels: df.loc[*row_lable*, *col_label*]

- Selection by Position using `DataFrame.iloc()` or `DataFrame.at()`

```
>>> df.iloc[3:5, 0:2]
```

	A	B
2013-01-04	-0.763038	0.477575
2013-01-05	0.621200	-0.618775

```
>>> df.iat[1, 1]
```

fast access to a scalar (equivalent to df.iloc[1, 1])

-0.528896863769626

By location: df.iloc[*row_loc*, *col_loc*]

Boolean Indexing

SUSTech Southern University of Science and Technology

- Using a single column's values to select data:

```
>>> df[df["A"] > 0]
```

	A	B	C	D
2013-01-05	0.621200	-0.618775	1.692731	-0.037749
2013-01-06	0.259993	-0.812927	-0.962159	-0.155331

```
>>> df.loc[df["A"] > 0]
```

	A	B	C	D
2013-01-05	0.621200	-0.618775	1.692731	-0.037749
2013-01-06	0.259993	-0.812927	-0.962159	-0.155331

- Selecting values from a `DataFrame` where a boolean condition is met:

```
>>> df[df > 0]
```

	A	B	C	D
2013-01-01	NaN	0.400826	NaN	0.782346
2013-01-02	NaN	NaN	0.940951	NaN
2013-01-03	NaN	1.120746	NaN	0.474923
2013-01-04	NaN	0.477575	NaN	NaN
2013-01-05	0.621200	NaN	1.692731	NaN
2013-01-06	0.259993	NaN	NaN	NaN

- Using the `isin()` method for filtering:

```
>>> df2 = df.copy()
>>> df2["E"] = ["one", "one", "two", "three", "four", "three"]
>>> df2.head()
```

	A	B	C	D	E
2013-01-01	-0.023244	0.400826	-1.392608	0.782346	one
2013-01-02	-1.437795	-0.528897	0.940951	-1.840915	one
2013-01-03	-0.986200	1.120746	-0.974936	0.474923	two
2013-01-04	-0.763038	0.477575	-0.585073	-1.098130	three
2013-01-05	0.621200	-0.618775	1.692731	-0.037749	four
2013-01-06	0.259993	-0.812927	-0.962159	-0.155331	three

```
>>> df2[df2["E"].isin(["two", "four"])]
```

	A	B	C	D	E
2013-01-03	-0.9862	1.120746	-0.974936	0.474923	two
2013-01-05	0.6212	-0.618775	1.692731	-0.037749	four

Dealing with NA

- `pandas` primarily uses the value `np.nan` to represent missing data. It is by default not included in computations.

- `isna()` gets the boolean mask where values are nan:

```
>>> df.loc[["20130101", "20130106"], ['B', 'D']] = np.nan
>>> df
```

	A	B	C	D
2013-01-01	-0.023244	NaN	-1.392608	NaN
2013-01-02	-1.437795	-0.528897	0.940951	-1.840915
2013-01-03	-0.986200	1.120746	-0.974936	0.474923
2013-01-04	-0.763038	0.477575	-0.585073	-1.098130
2013-01-05	0.621200	-0.618775	1.692731	-0.037749
2013-01-06	0.259993	NaN	-0.962159	NaN

```
>>> pd.isna(df)
```

	A	B	C	D
2013-01-01	False	True	False	True
2013-01-02	False	False	False	False
2013-01-03	False	False	False	False
2013-01-04	False	False	False	False
2013-01-05	False	False	False	False
2013-01-06	False	True	False	True

- `DataFrame.dropna()` drops any rows that have missing data:

```
>>> df.dropna(how="any")
```

	A	B	C	D
2013-01-02	-1.437795	-0.528897	0.940951	-1.840915
2013-01-03	-0.986200	1.120746	-0.974936	0.474923
2013-01-04	-0.763038	0.477575	-0.585073	-1.098130
2013-01-05	0.621200	-0.618775	1.692731	-0.037749

- `DataFrame.fillna()` fills missing data

```
>>> df.fillna(value=0.5)
```

	A	B	C	D
2013-01-01	-0.023244	0.500000	-1.392608	0.500000
2013-01-02	-1.437795	-0.528897	0.940951	-1.840915
2013-01-03	-0.986200	1.120746	-0.974936	0.474923
2013-01-04	-0.763038	0.477575	-0.585073	-1.098130
2013-01-05	0.621200	-0.618775	1.692731	-0.037749
2013-01-06	0.259993	0.500000	-0.962159	0.500000

concat DataFrames

SUSTech Southern University of Science and Technology

- Concatenating pandas objects together along an axis with concat():

- concat() across rows:

```
>>> df = pd.DataFrame(np.random.randn(10, 4))
>>> df.columns = ['A', 'B', 'C', 'D']
>>> df.head(2)
```

	A	B	C	D
0	0.596169	1.487679	2.369310	-1.255890
1	1.524799	0.632513	-2.103302	-0.304722

```
# break it into pieces
>>> pieces = [df[:3], df[3:7], df[7:]]
>>> pd.concat(pieces, axis=0)
```

	A	B	C	D
0	-2.783395	-0.210046	-0.569222	0.911045
1	0.381088	-1.226450	0.434870	-0.636089
2	1.117621	-0.892886	1.122575	-1.204585
3	0.097502	-0.696917	1.520378	0.286139
4	-0.226048	0.661643	-0.380691	0.734967
5	-0.145784	-2.020037	-1.435043	0.531568
6	0.531240	-1.430163	-0.766562	0.425568
7	0.788771	1.343522	-1.301957	-2.234962
8	1.263672	0.971128	1.173103	-0.111349
9	-0.149878	-0.221053	1.383188	1.919177

- concat() across columns:

```
>>> pd.concat([df[["A", "B"]], df.D], axis=1)
```

	A	B	D
0	-2.783395	-0.210046	0.911045
1	0.381088	-1.226450	-0.636089
2	1.117621	-0.892886	-1.204585
3	0.097502	-0.696917	0.286139
4	-0.226048	0.661643	0.734967
5	-0.145784	-2.020037	0.531568
6	0.531240	-1.430163	0.425568
7	0.788771	1.343522	-2.234962
8	1.263672	0.971128	-0.111349
9	-0.149878	-0.221053	1.919177

Adding a column to a DataFrame is relatively fast. However, adding a row requires a copy, and may be expensive. It's better to pass a pre-built list of records to the DataFrame constructor instead of building a DataFrame by iteratively appending records to it.

merge DataFrames

- `merge()` enables SQL style join types along specific columns.

```
>>> left = pd.DataFrame({"key": ["foo", "foo"],
                         "lval": [1, 2]})
>>> left
```

	key	lval
0	foo	1
1	foo	2

```
>>> right = pd.DataFrame({"key": ["foo", "foo"],
                          "rval": [4, 5]})
>>> right
```

	key	rval
0	foo	4
1	foo	5

```
>>> pd.merge(left, right, on="key")
```

	key	lval	rval
0	foo	1	4
1	foo	1	5
2	foo	2	4
3	foo	2	5

```
>>> left = pd.DataFrame({"key": ["foo", ”bar"],
                         "lval": [1, 2]})
>>> left
```

	key	lval
0	foo	1
1	bar	2

```
>>> right = pd.DataFrame({"key": ["foo", ”bar"],
                          "rval": [4, 5]})
>>> right
```

	key	rval
0	foo	4
1	bar	5

```
>>> pd.merge(left, right, on="key")
```

	key	lval	rval
0	foo	1	4
1	bar	2	5

Grouping

SUSTech Southern University of Science and Technology

- By "group by" we are referring to a process involving one or more of the following steps:
  - Splitting the data into groups based on some criteria
  - Applying a function to each group independently
  - Combining the results into a data structure

```
>>> df = pd.DataFrame(
...:     {
...:         "A": ["foo", "bar", "foo", "bar", "foo", "bar", "foo", "foo"],
...:         "B": ["one", "one", "two", "three", "two", "two", "one", "three"],
...:         "C": np.random.randn(8),
...:         "D": np.random.randn(8),
...:     })
>>> df
```

	A	B	C	D
0	foo	one	0.526672	0.211162
1	bar	one	0.173828	-0.144316
2	foo	two	-0.773200	0.965731
3	bar	three	-0.126775	0.833290
4	foo	two	0.535224	1.559121
5	bar	two	-0.233228	0.889734
6	foo	one	0.023856	-0.305266
7	foo	three	-0.730864	0.338401

```
>>> df.groupby("A")[["C", "D"]].sum()
```

A	C	D
bar	-0.186175	1.578709
foo	-0.418312	2.769149

```
>>> df.groupby(["A", "B"]).sum()
```

A	B	C	D
bar	one	0.173828	-0.144316
	three	-0.126775	0.833290
	two	-0.233228	0.889734
foo	one	0.550528	-0.094104
	three	-0.730864	0.338401
	two	-0.237976	2.524852

Categoricals

SUSTech Southern University of Science and Technology

- pandas can include categorical data in a DataFrame

```
>>> df = pd.DataFrame(
...: {"id": [1, 2, 3, 4, 5, 6],
...:  "raw_grade": ["a", "b", "b", "a", "a", "e"]}
...: )
# Converting the raw grades to a categorical data type:
>>> df["grade"] =  df["raw_grade"].astype("category")
>>> df.dtypes
>>> df
```

	data
id	int64
raw_grade	object
grade	category

	id	raw_grade	grade
0	1	a	a
1	2	b	b
2	3	b	b
3	4	a	a
4	5	a	a
5	6	e	e

```
# Rename the categories to more meaningful names:
>>> new_categories = ["very good", "good", "very bad"]
>>> df["grade"] =
 ...: df["grade"].cat.rename_categories(new_categories)
>>> df["grade"] = df["grade"].cat.set_categories(
...: ["very bad", "bad", "medium", "good", "very good"])
>>> df['grade']
```

```
0    very good
1         good
2         good
3    very good
4    very good
5     very bad
Name: grade, dtype: category
Categories (5, object): ['very bad', 'bad', 'medium', 'good', 'very good']
```

```
# Sorting is per order in the categories, not lexical order:
>>> df.sort_values(by="grade", ascending=True)
```

	id	raw_grade	grade
5	6	e	very bad
1	2	b	good
2	3	b	good
0	1	a	very good
3	4	a	very good
4	5	a	very good

Exercises

Exercise 1

- `Concat` the following DataFrames:

	col1	col2	col3
0	1	a	a1
1	2	b	b2
2	3	c	c3

	col1	col2	col3
0	4	d	d4
1	5	e	e5
2	6	f	f6

	col1	col2	col3
0	7	g	g7
1	8	h	h2
2	9	i	i3

	col1	col2	col3
0	1	a	a1
1	2	b	b2
2	3	c	c3
3	4	d	d4
4	5	e	e5
5	6	f	f6
6	7	g	g7
7	8	h	h2
8	9	i	i3

Exercise 2

- Sort the following DataFrame according to the `Year` categorical order:

  `['Freshman' < 'Sophomore' < 'Junior' < 'Senior']`

```
    Name      Year Shirt_Size
0    Tim    Junior          S
1  Sarah    Senior          M
2  Hasan  Freshman          L
3  Jyoti    Junior          S
4   Jack  Freshman          L
```

	Name	Year	Shirt_Size
2	Hasan	Freshman	L
4	Jack	Freshman	L
0	Tim	Junior	S
3	Jyoti	Junior	S
1	Sarah	Senior	M

Exercise 3

SUSTech Southern University of Science and Technology

- Calculate the mean device price for each company:

	price	brand	device
0	200	apple	phone
1	300	google	phone
2	400	apple	computer
3	500	apple	phone
4	300	google	computer
5	600	apple	computer
6	700	google	phone
7	900	google	computer

brand	device	price
apple	computer	500.0
	phone	350.0
google	computer	600.0
	phone	500.0