# Data Processing, Analysis and Visualization in Python

Pandas

1

## What is Pandas?

- pandas – provides the **DataFrame** object for efficient storage and manipulation

Series 1

| | Mango |
|---|---|
| 0 | 4 |
| 1 | 5 |
| 2 | 6 |
| 3 | 3 |
| 4 | 1 |

+

Series 2

| | Apple |
|---|---|
| 0 | 5 |
| 1 | 4 |
| 2 | 3 |
| 3 | 0 |
| 4 | 2 |

+

Series 3

| | Banana |
|---|---|
| 0 | 2 |
| 1 | 3 |
| 2 | 5 |
| 3 | 2 |
| 4 | 7 |

=

DataFrame

| | Mango | Apple | Banana |
|---|---|---|---|
| 0 | 4 | 5 | 2 |
| 1 | 5 | 4 | 3 |
| 2 | 6 | 3 | 5 |
| 3 | 3 | 0 | 2 |
| 4 | 1 | 2 | 7 |

Pandas

2

## What is Pandas?

- pandas – provides the **DataFrame** object for efficient storage and manipulation
- pandas is a newer package built on top of NumPy (and matplotlib)
- pandas provides an efficient implementation of a DataFrame
- DataFrames are essentially multidimensional arrays with attached row and column labels, and often with heterogeneous types and/or missing data
- pandas implements several powerful data operations (e.g., groupings, pivots) familiar to users of both database frameworks and spreadsheet programs of labeled/columnar data.

Pandas

3

## A simple pandas example

```
>>> import pandas as pd
>>> pd.__version__

>>> data = pd.Series([0.25, 0.5, 0.75, 1.0])
>>> data

>>> data.values

>>> type(data.values)

>>> data.index

>>> type(data.index)
```

4

## Series in pandas

- The pandas Series has an explicitly defined index associated with the values.
- A Series is a structure that maps typed keys to a set of typed values.
- Pandas Series is more efficient than Python dictionaries for certain operations.

```
>>> data = pd.Series([0.25, 0.5, 0.75, 1.0], index=['a', 'b',
'c', 'd'])
>>> data
```

```
>>> data['b']
```

5

## Dictionaries vs Pandas

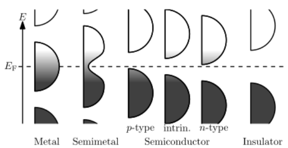
- Dictionaries are one of python's default data structures which allow you to store key: value pairs and offer some built-in methods to manipulate your data.
- Panda's Series are one-dimensional ndarrays with axis-labels, which allow you to store array-like, dict, or scalar values and are one of numpy's (a scientific computing python library) built-in data structures.

6

## Dictionaries vs Pandas

```
# Dict
>>> bandgap_dict = {'Hg0.7Cd0.3Te':0.35, 'CuBr':3.08,
'LuP':1.30, 'Cu3SbSe4':0.40, 'ZnO':3.44}
>>> bandgap_dict
```

```
# Pandas
>>> bandgaps = pd.Series(bandgap_dict)
>>> bandgaps
```




7

## Accessing data with Pandas

```
>>> bandgaps['CuBr']

>>> bandgaps['Hg0.7Cd0.3Te':'ZnO']
```

8

# Series creation

- pd.Series(data, index=index)

```
>>> pd.Series([2, 4, 6]) # list or NumPy array
```

```
>>> pd.Series(5, index=[100, 200, 300]) # repeated scalar
```

9

# Series creation

- pd.Series(data, index=index)

```
>>> pd.Series({2:'a', 1:'b', 3:'c'}) # dictionary
```

```
>>> pd.Series({2:'a', 1:'b', 3:'c'}, index=[3, 2])
```

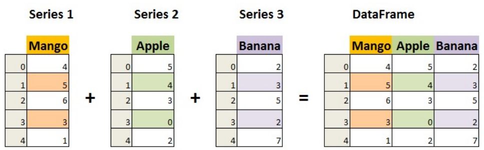
```
>>> l = pd.Series({2:'a', 1:'b', 3:'c'}, index=[3, 1, 2])
```

- Sort l according to index and values


10

# Dataframes

- DataFrame can be thought of either as a generalization of a NumPy array, or as a specialization of a Python dictionary.
- If a Series is an analog of a one-dimensional array with flexible indices, a DataFrame is an analog of a two-dimensional array with both flexible row indices and flexible column names.
- A DataFrame is a sequence of aligned Series objects, which share the same index.




11

# Dataframes (from Series)

```
>>> bandgap_dict = {'Hg0.7Cd0.3Te':0.35, 'CuBr':3.08,
'LuP':1.30, 'Cu3SbSe4':0.40, 'ZnO':3.44}
>>> bandgaps = pd.Series(bandgap_dict); bandgaps
```

```
>>> is_metal_dict = {'Hg0.7Cd0.3Te':'', 'CuBr':False,
'LuP':False, 'Cu3SbSe4':False, 'ZnO':False}
>>> is_metal = pd.Series(is_metal_dict); is_metal
```

12

## Dataframes (from Series)

```
>>> materials = pd.DataFrame({'bandgap':bandgaps,
'is_metal':is_metal})
>>> materials
```

13

## Dataframes

```
>>> materials.index

>>> materials.columns

>>> materials['bandgap']
```

14

## Dataframe creation

- From a single Series object

```
>>> bandgaps    # Series

>>> type(bandgaps)

>>> dataframe = pd.DataFrame(bandgaps, columns=['bandgap'])
>>> type(dataframe)

>>> dataframe    # DataFrame
```

15

## Dataframe creation

- From list of dicts

```
>>> data = [{'a': i, 'b': 2 * i} for i in range(3)]
>>> data

>>> pd.DataFrame(data)
```

- From a dictionary of Series objects (we already saw this)

```
>>> materials = pd.DataFrame({'bandgap':bandgaps,
'is_metal':is_metal})
```

16

## Dataframe creation

- `Create a DataFrame from` a two-dimensional NumPy array of random numbers. Use column and index labels.


17

## Inspecting a dataframe

```
>>> materials
```

| | bandgap | is_metal |
|---|---|---|
| Hg0.7Cd0.3Te | 0.35 | |
| CuBr | 3.08 | False |
| LuP | 1.30 | False |
| Cu3SbSe4 | 0.40 | False |
| ZnO | 3.44 | False |

```
>>> materials.head(2)

>>> materials.tail(2)
```

Try these functions as well:
`df.head(), df.tail(),`
`df.info(), df.describe()`
`df.index, df.columns,`
`df.values, df.shape`

What do the commands do?
What are their types?


18

## Index

- The Series and DataFrame objects contain an explicit index that lets you reference and modify data.
- The Index object can be thought of either as an *immutable array* or as an *ordered set* (technically a *multiset*, as Index objects may contain repeated values).

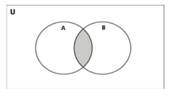
```
>>> ind = pd.Index([2, 3, 5, 7, 11])
>>> ind

>>> ind[1]

>>> ind[::2]

>>> print(ind.size, ind.shape, ind.ndim, ind.dtype)

>>> ind[1] = 0
```

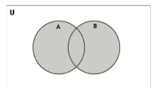
19

## Index

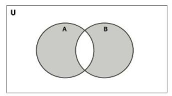
- The Series and DataFrame objects contain an explicit index that lets you reference and modify data.
- The Index object can be thought of either as an *immutable array* or as an *ordered set* (technically a *multiset*, as Index objects may contain repeated values).

```
>>> indA = pd.Index([1, 3, 5, 7, 9])
>>> indB = pd.Index([2, 3, 5, 7, 11])
>>> indA & indB    # Intersection
```





```
>>> indA | indB    # Union
```



```
>>> indA ^ indB    # Symmetric difference
```

20

## Data indexing and selection

- Recall: access, set, and modify values in NumPy arrays:

```
>>> import numpy.random as rnd
>>> arr = rnd.random((5,5))
>>> arr[2, 1]         # indexing
>>> arr[:, 1:2]       # slicing
>>> arr[arr > 0.5]    # masking
>>> arr[0, [1, 2]]    # fancy indexing
>>> arr[:, [1, 2]]    # combinations
```


21

## Data selection in Series (as Dicts)

- Recall: access, set, and modify values in NumPy arrays:

```
>>> data = pd.Series([0.25, 0.5, 0.75, 1.0], index=['a', 'b',
'c', 'd']); data

>>> data['b']

>>> 'a' in data

>>> data.keys() # Try data.values
```

22

## Data selection in Series (as 1D arrays)

```
>>> data['a':'c'] # slicing by explicit index

>>> data[0:2] # slicing by implicit integer index

>>> data[(data > 0.3) & (data < 0.8)] # masking
```

23

## Data selection in Series (explicit vs. implicit indexing)

```
>>> data = pd.Series(['a', 'b', 'c'], index=[1, 3, 5]); data

>>> data[1]    # explicit index (i.e., key) when indexing

>>> data[1:3] # implicit index (i.e., position) when slicing
```

24

## Data selection in Series (loc vs. iloc)

```
>>> data = pd.Series(['a', 'b', 'c'], index=[1, 3, 5])
```

- loc – indexing and slicing that references the explicit index

```
>>> data.loc[1]
```

Index by *key*

```
>>> data.loc[1:3]
```

- iloc – indexing and slicing that references the implicit index

```
>>> data.iloc[1]
```

Index by *array location*

```
>>> data.iloc[1:3]
```

25

## Data selection in DataFrames

```
>>> df = pd.DataFrame(rng.integers(0, 10, (3, 3)), columns=list('ABC'), index=list('ijk'))
>>> df
>>> df.values     # NumPy array
>>> df.values[0] # Row
>>> df.values[0, 1] # df.values[0][1] is the same
>>> df.iloc[0, 1]    # df.iloc[0][1] is the same
>>> df.loc['i']['A'] # df.loc['i', 'A'] is the same
>>> df.loc[df.A > 3] # Checks element, prints row
>>> df['A']      # Column
>>> df['A'][1] # df['A'].iloc[1] is the same
>>> df['i':'j']
>>> df['i':'k']['A']
>>> df['A']['i']      # Try type(df['A']['i'])
>>> df['i':'i']['A'] # Try type(df['i':'i']['A'])
>>> df['i']['A']      # Will this work???
```

| | A | B | C |
|---|---|---|---|
| i | 3 | 8 | 3 |
| j | 5 | 5 | 2 |
| k | 1 | 2 | 6 |


26

## Data selection in Series

```
>>> materials = pd.DataFrame({'bandgap':bandgaps, 'is_metal':is_metal}); materials
```

```
>>> materials.bandgap
Hg0.7Cd0.3Te     0.35
CuBr             3.08
LuP              1.30
Cu3SbSe4         0.40
ZnO              3.44
Name: bandgap, dtype: float64
```

```
>>> materials['bandgap']
Hg0.7Cd0.3Te     0.35
CuBr             3.08
LuP              1.30
Cu3SbSe4         0.40
ZnO              3.44
Name: bandgap, dtype: float64
```

```
>>> materials.bandgap == materials['bandgap']
```

27

## Data selection in Series

- Dictionary-style syntax can be used to modify the object, in this case to modify a column:

```
>>> materials['is_metal'] = materials['bandgap'] < 0.05
>>> materials['is_metal']
```

```
>>> materials['is_metal'][0] = False    # Modifies specific value
```

28

## Data selection in Series

- Dictionary-style syntax can be used to modify the object, in this case to add a new column:

```
>>> materials['is_insulator'] = materials['bandgap'] > 4.0
>>> materials['is_insulator']
```

29

## Data selection in Series

- `Our updated DataFrame is now:`

```
>>> materials
```

30

## Data selection in DataFrames

- DataFrame as 2D array:

```
>>> materials.values    # Array of array

>>> materials.T    # What does this do?
```

31

## Data selection in DataFrames

- DataFrame as 2D array:

```
>>> materials.values[0]

>>> materials.values[0][0]

>>> materials.values[1][2]

>>> materials.values[1][3]    # What happens here?
```

32

## Data selection in DataFrames

- DataFrame as 2D array and dict:

```
>>> materials
              bandgap  is_metal  is_insulator
Hg0.7Cd0.3Te     0.35     False         False
CuBr             3.08     False         False
LuP              1.30     False         False
Cu3SbSe4         0.40     False         False
ZnO              3.44     False         False
>>> materials.values[1][0]   # 2D array

>>> materials['bandgap']['CuBr'] # dict

>>> materials['bandgap'].values[1] # mixed
```

33

## Data selection in DataFrames

- Data selection in DataFrame – loc and iloc

```
>>> materials.loc[materials.bandgap > 2.0]

>>> materials.loc[materials.bandgap > 2.0, ['bandgap']] # By column

>>> materials.loc[materials.bandgap > 2.0, ['is_metal']] # By column
```

34

## Data selection in DataFrames

- While indexing refers to columns, slicing refers to rows
- Slices can also refer to rows by number rather than by index

```
>>> materials
              bandgap  is_metal  is_insulator
Hg0.7Cd0.3Te     0.35     False         False
CuBr             3.08     False         False
LuP              1.30     False         False
Cu3SbSe4         0.40     False         False
ZnO              3.44     False         False
>>> materials['CuBr':'LuP']    # materials[1:3]
```

35

## √ Data selection in DataFrames

- While indexing refers to columns, slicing refers to rows
- Masking operations are also interpreted row-wise rather than column-wise

```
>>> materials[materials.bandgap > 2.0]
              bandgap  is_metal  is_insulator
Hg0.7Cd0.3Te     0.35     False         False
CuBr             3.08     False         False
LuP              1.30     False         False
Cu3SbSe4         0.40     False         False
ZnO              3.44     False         False
>>> materials[materials.bandgap > 2.0]
```

36

# Operating on Data in Pandas

- NumPy performs quick element-wise operations, both with
  - basic arithmetic – addition, subtraction, multiplication, etc., and
  - sophisticated operations – trigonometric, exponential, logarithmic functions

  Pandas inherits much of this functionality and the ufuncs from NumPy
- Pandas includes a couple of useful twists:
  - For **unary** operations like negation and trigonometric functions, these ufuncs will **preserve index and column labels** in the output
  - For **binary** operations such as addition and multiplication, Pandas will automatically **align indices** when passing the objects to the ufunc

  This means that keeping the context of data and combining data from different sources—both potentially error-prone tasks with raw NumPy arrays—become essentially foolproof ones with Pandas.
- There are well-defined operations between one-dimensional Series structures and two-dimensional DataFrame structures

37

# Ufuncs: Index preservation

```
>>> rng = np.random.default_rng(12345)
>>> rints = rng.integers(0, 10, 4)
>>> pd.Series(rints)
```

```
>>> rng = np.random.default_rng(12345)
>>> rints = rng.integers(0, 10, (3, 4))
>>> df = pd.DataFrame(rints, columns=['A', 'B', 'C', 'D'])
>>> df
```

38

# Ufuncs: Index alignment in series

```
>>> A = pd.Series([2, 4, 6], index=[0, 1, 2])
>>> B = pd.Series([1, 3, 5], index=[1, 2, 3])
>>> A + B    # A.add(B) is the same
```

A

| | |
|---|---|
| 0 | 2 |
| 1 | 4 |
| 2 | 6 |

B

| | |
|---|---|
| 1 | 1 |
| 2 | 3 |
| 3 | 5 |

```
>>> A.add(B, fill_value=0)
```

39

# Ufuncs: Index alignment in series

- Alignment takes place for both columns and indices; indices in the result are sorted.

```
>>> A = pd.DataFrame(rng.integers(0, 20, (2, 2)),
columns=list('AB'))
>>> B = pd.DataFrame(rng.integers(0, 10, (3, 3)),
columns=list('BAC'))
```

40

## Ufuncs: Index alignment in series

- Alignment takes place for both columns and indices; indices in the result are sorted.

```
>>> fill = A.stack().mean()
>>> A.add(B, fill_value=fill)
```

```
>>> A
    A   B
0   4  13
1  12  18
```

```
>>> B
   B  A  C
0  7  2  9
1  9  7  6
2  1  0  2
```

What's the meaning of
stack()
mean()
???


41

## Ufuncs: Index alignment in series

- Additional methods

| Python operator | Pandas method(s) |
|---|---|
| + | add() |
| - | sub(), subtract() |
| * | mul(), multiply() |
| / | truediv(), div(), divide() |
| // | floordiv() |
| % | mod() |
| ** | pow() |

42

## Operations between DataFrame and Series

```
>>> A = np.array([[3, 8, 2, 4], [2, 6, 4, 8], [6, 1, 3, 8]])
>>> df = pd.DataFrame(A, columns=list('ijkl'))
>>> df
```

43

## Operations between DataFrame and Series

```
>>> df - df.iloc[0]  # Default operation is row-wise
```

```
>>> df - df.iloc[0][0] # Same as df - df.iloc[0, 0]
```

df

```
   i  j  k  l
0  3  8  2  4
1  2  6  4  8
2  6  1  3  8
```

```
>>> df.subtract(df['i'], axis=0) # Operation can be column-wise
```

What will
df.subtract(df.iloc[0],axis=1)
do???

44

## Operations between DataFrame and Series

```
>>> half_row = df.iloc[0, ::2]
>>> half_row
```

df

| | i | j | k | l |
|---|---|---|---|---|
| 0 | 3 | 8 | 2 | 4 |
| 1 | 2 | 6 | 4 | 8 |
| 2 | 6 | 1 | 3 | 8 |

```
>>> df - half_row # Remember: row-wise
```

45

## Handling missing data

- Real-world data is rarely clean and homogeneous.
- Many interesting datasets will have some amount of data missing.
- Different data sources may indicate missing data in different ways.

46

## Missing data conventions

- In the **masking** approach, the mask might be an entirely separate `bool` array.
  - A separate mask array requires allocation of an additional bool array, which adds overhead in both storage and computation.
- In the **sentinel** approach, the sentinel value might be some data-specific convention, –9999, NaN or some rare bit pattern.
  - A sentinel value reduces the range of valid values that can be represented and may require extra logic in CPU and GPU arithmetic.
  - Common special values like NaN are not available for all data types.
- Pandas use sentinels for missing data (the two already-existing Python null values):
  - the special floating point **NaN** value
  - the Python **None** object

47

## None: Pythonic missing data

- Because None is a Python object, it cannot be used in any arbitrary NumPy/Pandas array, but only in arrays with data type **object**

```
>>> vals1 = np.array([1, None, 3, 4])
>>> vals1
```

- **Slow** – any operations on the data will be done at the Python level, with much more overhead than the typically fast operations seen for arrays with native types.
- Aggregations like `sum()` or `min()` across an array with None value will generate an **error**

```
>>> vals1.sum()
```

48

## NaN: Missing numerical data

- NaN (Not a Number) is a special floating-point value recognized by all systems that use the standard IEEE floating-point representation

```
>>> vals2 = np.array([1, np.nan, 3, 4]); vals2.dtype
```

- NumPy chose a native floating-point type for this array
- This array supports fast operations pushed into compiled code
- The result of arithmetic with NaN will be another NaN

```
>>> 1 + np.nan

>>> 8 * np.nan
```

49

## NaN: Missing numerical data

- Aggregates over the values are well defined (i.e., they don't result in an error) but not always useful:

```
>>> vals2.sum(), vals2.min(), vals2.max()
```

- NumPy provides special aggregations that ignore these missing values:

```
>>> np.nansum(vals2), np.nanmin(vals2), np.nanmax(vals2)
```

- **NaN is specifically a floating-point value!**
  - There is no equivalent NaN value for integers, strings, or other types.

50

## NaN and None

- Pandas convert between NaN and None where appropriate

```
>>> pd.Series([1, np.nan, 2, None])
```

If we set a value in an integer array to **None**, it will automatically be cast to a **floating-point**

```
>>> x = pd.Series(range(2), dtype=int); x

>>> x[0] = None; x
```

| Typeclass | Conversion when storing NAs | NA sentinel value |
|---|---|---|
| floating | No change | np.nan |
| object | No change | None or np.nan |
| integer | Cast to float64 | np.nan |
| boolean | Cast to object | None or np.nan |

51

## Operating on null values

- `isnull()`
  - Generate a Boolean mask indicating missing values
- `notnull()`
  - Opposite of `isnull()`
- `dropna()`
  - Return a filtered version of the data
- `fillna()`
  - Return a copy of the data with missing values filled or imputed

52

# Operating on null values
## isnull() and notnull()

```
>>> data = pd.Series([1, np.nan, 'hello', None])
>>> data
```

```
>>> data.notnull()
```

```
>>> data.isnull()
```

```
>>> data[data.notnull()] # Bool mask
```

53

# Operating on null values
## dropna() parameters

- The default is how='any', such that any row or column (depending on the axis keyword) containing a null value will be dropped. how='all' will only drop rows/columns that are all null values.

```
>>> df.dropna(axis='columns', how='any')
```

```
>>> df.dropna(axis='columns', how='all')
```

```
>>> df.dropna(axis='rows', how='any')
```

```
>>> df
     0    1  2
0  1.0  NaN  2
1  2.0  3.0  5
2  NaN  4.0  6
```

54

# Filling null values – fillna()

```
>>> data = pd.Series([1, np.nan, 2, None, 3], index=list('abcde'))
>>> data
```

```
>>> data.fillna(0)
```

55

# Filling null values – ffillna()

```
>>> data.fillna(method='ffill')   # forward-fill
```

```
>>> data.fillna(method='bfill')   # backward-fill
```

```
a    1.0
b    NaN
c    2.0
d    NaN
e    3.0
dtype: float64
```

56

## Simple Aggregations (Series)

```
>>> ds = data.fillna(method='ffill')   # forward-fill
>>> dsd = ds.describe()
>>> dsd
```

```
>>> dsd['count']
```

```
a    1.0
b    1.0
c    2.0
d    2.0
e    3.0
dtype: float64
```

57

## Simple Aggregations (DataFrames)

```
>>> materials
```

```
>>> materials.describe()
```

What about
materials.sum()
materials.mean()
materials.std()

58

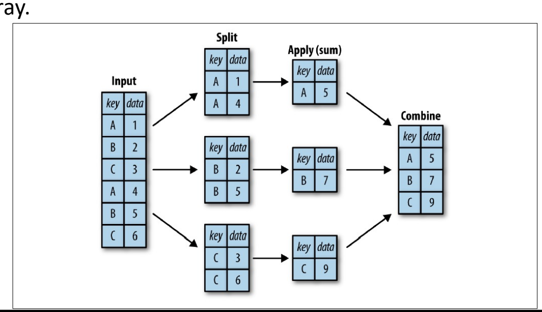
## groupby()

- Often, we would like to aggregate conditionally on some label or index
- The name "group by" comes from a command in the SQL
- Useful to think of it in the terms coined by Hadley Wickham: *split, apply, combine*
  - The **split** step involves breaking up and grouping a DataFrame depending on the value of the specified key.
  - The **apply** step involves computing some function, usually an aggregate, transformation, or filtering, within the individual groups.
  - The **combine** step merges the results of these operations into an output array.

59

## groupby()

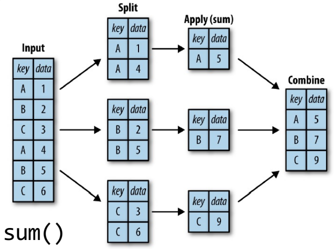
- **Split** breaks up and groups a DataFrame depending on the value of the specified key.
- **Apply** involves computing an aggregate, within the individual groups.
- **Combine** merges the results of these operations into an output array.




60

# groupby()

```
>>> df =
pd.DataFrame({'key':['A','B','C','A','B','C'],
'data':[1,2,3,4,5,6]})
>>> df
```

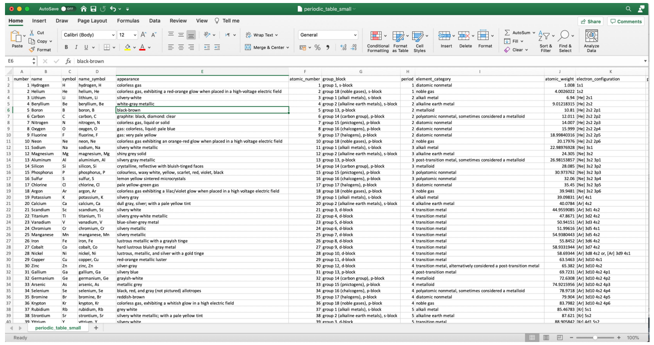



```
>>> df.groupby('key').sum()
```

61

# groupby()

- We'll now use the file for PT (under 'Data Sets'): `'periodic_table_small.csv'`



62

# groupby()

- Single command:

```
>>> # Melting points grouped by rows of PT:
>>> df.groupby('period')['melting_point'].mean()
period
1        7.470000
2      651.171448
3      651.169128
4     1225.649658
5     1387.564941
6     1522.220703
7     1272.517700
Name: melting_point, dtype: float32
```

64

# groupby()

- Multiple grouped summaries:

```
>>> # Summary for each row:
>>> df.groupby('period')['melting_point'].agg([min, max, sum])
               min      max           sum
period
1         0.950000    13.99     14.940000
2        24.559999  2349.00   4558.200195
3        83.809998  1687.00   4558.184082
4       115.779999  2183.00  20836.044922
5       161.399994  2896.00  24976.167969
6       202.000000  3695.00  48711.062500
7       340.000000  2023.00  21632.800781
```

65

## groupby()

- Grouping by multiple variables (multi-index):

```
>>> # Summary for each row:
>>> df.groupby(['period', 'phase'])['melting_point'].mean()
period  phase
1       gas               7.470000
2       gas              48.887501
        solid          1454.216675
3       gas             127.705002
        solid           860.554810
4       gas             115.779999
        liquid          265.799988
        solid          1363.630981
        …
```

What can we conclude from this?

66

## groupby()

- Many groups, many summaries:

```
>>> # Summary for each row:
>>> df.groupby(['period', 'phase'])['melting_point',
                                    'boiling_point'].mean()
                melting_point  boiling_point
period phase
1      gas           7.470000      12.246500
2      gas          48.887501      69.919250
       solid      1454.216675    2848.333252
3      gas         127.705002     163.205994
       solid       860.554810    1903.578003
4      gas         115.779999     119.930000
       liquid      265.799988     332.000000
       solid      1363.630981    2567.000000
…
```

What can we conclude from this?

67

## groupby()

- Try groupby() using other columns


68

## pivot_table()

- We'll now use the file for PT:

```
>>> df.pivot_table(values='melting_point', index='period')

        melting_point
period
1            7.470000
2          651.171448
3          651.169128
4         1225.649658
5         1387.564941
6         1522.220703
7         1272.517700
>>> pt = df.pivot_table(values='melting_point', index='period',
                        fill_value=0.0)
```

70

## pivot_table()

- We'll now use the file for PT:

```
>>> df.pivot_table(values='melting_point', index='period',
                   aggfunc=[np.mean, np.std], fill_value=0.0)
                 mean            std
        melting_point melting_point
period
1            7.470000      9.220672
2          651.171448    930.898845
3          651.169128    566.569954
4         1225.649658    720.435600
5         1387.564941    943.652274
6         1522.220703    959.577342
7         1272.517700    477.194378
```

71

## pivot_table()

- We'll now use the file for PT:

```
>>> df.pivot_table(values='melting_point', index='period',
              columns='phase', fill_value=0.0, margins=True)
phase          gas      liquid        solid  solid (predicted)          All
period
1         7.470000    0.000000     0.000000           0.000000     7.470000
2        48.887501    0.000000  1454.216675           0.000000   651.171448
3       127.705002    0.000000   860.554810           0.000000   651.169128
4       115.779999  265.799988  1363.630981           0.000000  1225.649658
5       161.399994    0.000000  1387.985596           0.000000  1315.833496
6       202.000000  234.320999  1609.157959           0.000000  1522.220703
7         0.000000    0.000000  1402.280029        1087.142822  1272.517700
All      85.916367  250.060486  1438.293579        1087.142822  1239.196045
```

72

## pivot_table()

- We'll now use the file for PT:

```
>>> pt = df.pivot_table(values=['melting_point',
        'boiling_point'], index='period', columns='phase',
        aggfunc={'melting_point':np.min, 'boiling_point':np.max},
        fill_value=0.0)
>>> # Write to csv
pt.to_csv('/Users/majort/Documents/Research/BIU/Courses/Python-
data processing, analysis and visualization/Data Sets/periodic
table/periodic_table_melt_boil.csv')
```

73

## pivot_table()

- Available functions with pivot_table:

| Function Name | NaN-safe Version | Description |
|---|---|---|
| np.sum | np.nansum | Compute sum of elements |
| np.prod | np.nanprod | Compute product of elements |
| np.mean | np.nanmean | Compute mean of elements |
| np.std | np.nanstd | Compute standard deviation |
| np.var | np.nanvar | Compute variance |
| np.min | np.nanmin | Find minimum value |
| np.max | np.nanmax | Find maximum value |
| np.argmin | np.nanargmin | Find index of minimum value |
| np.argmax | np.nanargmax | Find index of maximum value |
| np.median | np.nanmedian | Compute median of elements |
| np.percentile | np.nanpercentile | Compute rank-based statistics of elements |
| np.any | N/A | Evaluate whether any elements are true |
| np.all | N/A | Evaluate whether all elements are true |

74

## Summary

- Pandas provide useful data structures for use in Python
- Pandas is built on top of NumPy and inherits certain properties from NumPy
- groupby() and pivot_table() provide powerful functions to organize and manipulate data

75

76