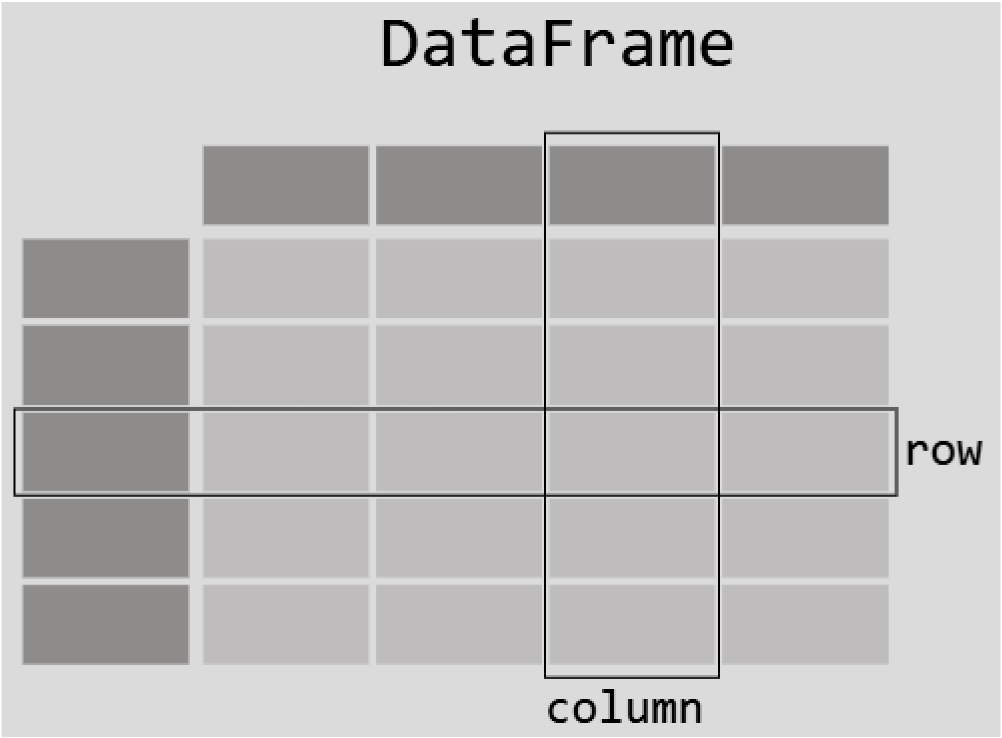
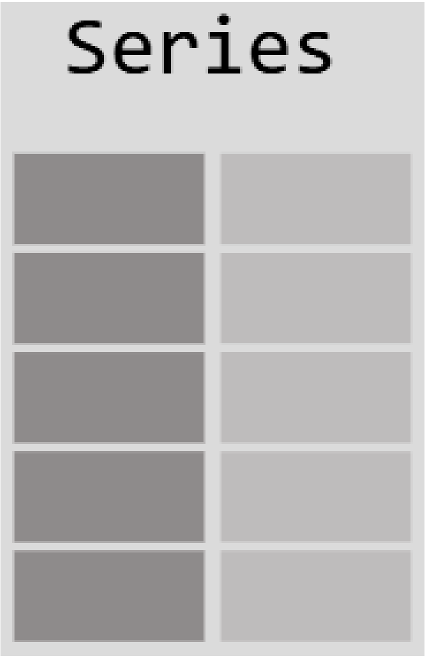
(U4284) Python程式設計
Package Intro - Pandas

Speaker: 吳淳硯

Enhanced NumPy?

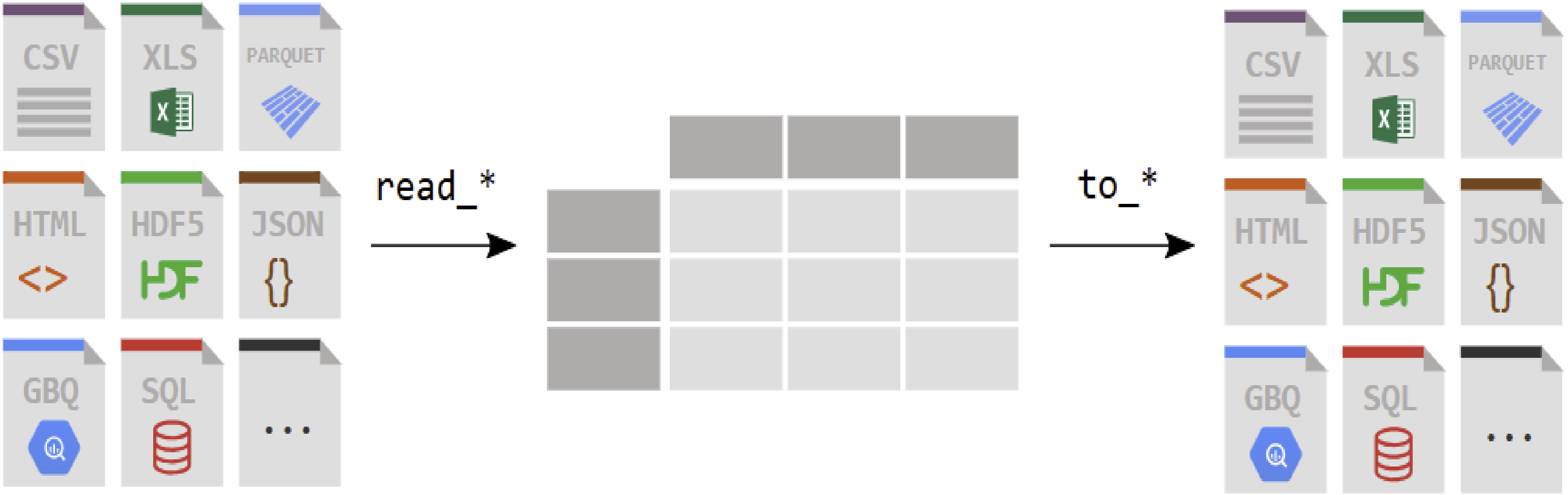
- Pandas, and in particular its *Series* and *DataFrame* objects, builds on the NumPy array structure and provides efficient access to these sorts of "data munging" tasks that occupy much of time.
- *DataFrames* are essentially 2-dim arrays with attached row and column labels, often with heterogeneous types and/or missing data. Each column in a *DataFrame* is a *Series*.





Intro

- How do I read and write tabular data?
  - Getting data in to pandas from many different file formats or data sources is supported by read_* functions.
  - Exporting data out of pandas is provided by different to_*methods.
- The head/tail/info methods and the dtypes attribute are convenient for a first check.



Pandas Series - 0

- A Pandas *Series* is a 1-dim array of indexed data. It can be created from a list or array. Like with NumPy array, data can be accessed by the associated index via bracket notation.
- Series can be viewed as Generalized NumPy array. NumPy array has an *implicitly defined* integer index, the Pandas *Series* has an *explicitly defined* index. This explicit index definition gives the *Series* object additional capabilities.

```
In [7]: data = pd.Series([0.25, 0.5, 0.75, 1.0],
                         index=['a', 'b', 'c', 'd'])
        data
Out[7]: a    0.25
        b    0.50
        c    0.75
        d    1.00
        dtype: float64
```

- Or noncontiguous or nonsequential indices

```
In [9]: data = pd.Series([0.25, 0.5, 0.75, 1.0],
                         index=[2, 5, 3, 7])
        data
Out[9]: 2    0.25
        5    0.50
        3    0.75
```

Pandas Series - 1

- Pandas *Series* a bit like a specialization of Python dictionary.

```
In [11]: population_dict = {'California': 39538223, 'Texas': 29145505,
                            'Florida': 21538187, 'New York': 20201249,
                            'Pennsylvania': 13002700}
         population = pd.Series(population_dict)
         population
Out[11]: California      39538223
         Texas           29145505
         Florida         21538187
         New York        20201249
         Pennsylvania    13002700
         dtype: int64
```

- Unlick a dictionary, thought, the *Series* also supports array-style operation such as slicing

```
In [12]: population['California']
Out[12]: 39538223
```

```
In [13]: population['California':'Florida']
Out[13]: California    39538223
         Texas         29145505
         Florida       21538187
         dtype: int64
```

Pandas DataFrame - 0

- *DataFrame* is an analog of a 2-dim array as an ordered sequence of aligned 1-dim columns.

```
Out[19]:              population    area
         California     39538223  423967
         Texas          29145505  695662
         Florida        21538187  170312
         New York       20201249  141297
         Pennsylvania   13002700  119280
```

- Like the *Series*, the *DataFrame* has an index attribute. Additionally the *DataFrame* has a *columns* attribute.

```
In [20]: states.index
Out[20]: Index(['California', 'Texas', 'Florida', 'New York', 'Pennsylvania'],
         > dtype='object')
```

```
In [21]: states.columns
Out[21]: Index(['population', 'area'], dtype='object')
```

- Thus the *DataFrame* can be thought of as a generalization of a 2-dim NumPy array.

Pandas DataFrame - 1

- Similarly, we can think of a *DataFrame* as a specialization of a dictionary. Where a dictionary maps a key to a value, a *DataFrame* maps a column name to a *Series* of column data.

```
In [22]: states['area']
Out[22]: California      423967
         Texas           695662
         Florida         170312
         New York        141297
         Pennsylvania    119280
         Name: area, dtype: int64
```

- *Series* and *DataFrame* both contain an *explicit index* that lets you reference and modify data. This *index* object can be thought of either as an *immutable array*.

```
In [30]: ind = pd.Index([2, 3, 5, 7, 11])
         ind
Out[30]: Int64Index([2, 3, 5, 7, 11], dtype='int64')
```

```
In [34]: ind[1] = 0
TypeError: Index does not support mutable operations
```

National Taipei University

How to select subset?

- When selecting subsets, square brackets [] are used. Inside these brackets, you can use a single column/row label, a list of column/row labels, a slice of labels, a conditional expression or a colon.
  - Indexing – arr[2,1]
  - Slicing – arr[:,1:5]
  - Masking – arr[arr > 0]
  - fancy indexing – arr[0,[1,5]]
  - combinations thereof – arr[:,[1,5]]

- The corresponding patterns in Pandas will feel very familiar, though there are a few quirks to be aware of.

Indexers - 0

- If a *Series* has an explicit integer index, an indexing operation such df[1] will use the explicit indices, which a slicing operation like df[1:3] will use the implicit Python-style indices.

```
Out[11]: 1    a
         3    b
         5    c
         dtype: object

In [12]: # explicit index when indexing
         data[1]
Out[12]: 'a'

In [13]: # implicit index when slicing
         data[1:3]
Out[13]: 3    b
         5    c
         dtype: object
```

- Because of this potential confusion. Pandas provides some special indexer attributes that explicitly expose certain indexing schemes – *loc* and *iloc* attribute.

Indexers - 1

- One guiding principle of Python code is that "explicit is better than implicit." The explicit nature of *loc* and *iloc* makes them helpful in maintaining clean and readable code.
  - *loc attribute*: allowing indexing and slicing that always references the explicit index.
  - *iloc attribute*: allows indexing and slicing that always references the implicit Python-style index.
- Columns of *DataFrame* can be accessed via dictionary-style indexing of the columns name. Equivalently, we can use attribute-style access with column names that are strings.

```
In [19]: data['area']
Out[19]: California      423967
         Texas           695662
         Florida         170312
         New York        141297
         Pennsylvania    119280
         Name: area, dtype: int64
```

```
In [20]: data.area
Out[20]: California      423967
         Texas           695662
         Florida         170312
         New York        141297
         Pennsylvania    119280
         Name: area, dtype: int64
```

Indexers - 2

- Attribute-style does not work for all case. If the column names are not strings, or if the column names conflict with methods of the *DataFrame*, this attribute-style access is not possible.

```
In [21]: data.pop is data["pop"]
Out[21]: False
```

- It should avoid the temptation to try column assignment via attributes. Use df[‘pop’] = 3 rather than data.pop = 3.

- Dictionary-style syntax can also be used to modify the object, in this case adding a new column.

```
In [22]: data['density'] = data['pop'] / data['area']
         data
Out[22]:                 area       pop     density
         California    423967  39538223   93.257784
         Texas         695662  29145505   41.896072
         Florida       170312  21538187  126.463121
         New York      141297  20201249  142.970120
         Pennsylvania  119280  13002700  109.009893
```

DataFrame as 2-dim Array

- We can examine the raw underlying data array using the values attribute.

```
In [23]: data.values
Out[23]: array([[4.23967000e+05, 3.95382230e+07, 9.32577842e+01],
                [6.95662000e+05, 2.91455050e+07, 4.18960717e+01],
                [1.70312000e+05, 2.15381870e+07, 1.26463121e+02],
                [1.41297000e+05, 2.02012490e+07, 1.42970120e+02],
                [1.19280000e+05, 1.30027000e+07, 1.09009893e+02]])
```

- With this in mind, many array-like operations can be done on the *DataFrame* itself. In particular, passing a single index to an array accesses a row and passing a single index to a *DataFrame* access a column.

```
In [25]: data.values[0]
Out[25]: array([4.23967000e+05, 3.95382230e+07, 9.32577842e+01])

In [26]: data['area']
Out[26]: California      423967
         Texas           695662
         Florida         170312
         New York        141297
         Pennsylvania    119280
         Name: area, dtype: int64
```

loc and *iloc* indexer

- Using *loc* indexer we can index the underlying data in an array-like style but using *explicit* index and column names.

```
In [28]: data.loc[:'Florida', :'pop']
Out[28]:             area       pop
         California  423967  39538223
         Texas       695662  29145505
         Florida     170312  21538187
```

- Any of the familiar NumPy-style data access patterns can be used within these indexers.

```
In [29]: data.loc[data.density > 120, ['pop', 'density']]
Out[29]:                pop     density
         Florida   21538187  126.463121
         New York  20201249  142.970120
```

- Any of these indexing conventions may also be used to set or modify values.

```
In [30]: data.iloc[0, 2] = 90
         data
Out[30]:                area       pop     density
         California   423967  39538223   90.000000
         Texas        695662  29145505   41.896072
         Florida      170312  21538187  126.463121
         New York     141297  20201249  142.970120
         Pennsylvania 119280  13002700  109.009893
```

93.257784

Index Preserving

- If we apply a NumPy ufunc on either of these objects, the result will be another Pandas object with the indices preserved.

```
In [2]: rng = np.random.default_rng(42)
        ser = pd.Series(rng.integers(0, 10, 4))
        ser
Out[2]: 0    0
        1    7
        2    6
        3    4
        dtype: int64
```

```
In [4]: np.exp(ser)
Out[4]: 0       1.000000
        1    1096.633158
        2     403.428793
        3      54.598150
        dtype: float64
```

- This is true also for more involves sequence of operations.

```
In [3]: df = pd.DataFrame(rng.integers(0, 10, (3, 4)),
                          columns=['A', 'B', 'C', 'D'])
        df
Out[3]:    A  B  C  D
        0  4  8  0  6
        1  2  0  5  9
        2  7  7  7  7
In [5]: np.sin(df * np.pi / 4)
Out[5]:               A             B         C         D
        0  1.224647e-16 -2.449294e-16  0.000000 -1.000000
        1  1.000000e+00  0.000000e+00 -0.707107  0.707107
        2 -7.071068e-01 -7.071068e-01 -0.707107 -0.707107
```

Index Alignment

- For binary operations on two or objects, Pandas will align indices *Series* or *DataFrame* in the process of performing the operation. Any item for which one or the other does not have an entry is marked with NaN (Not a Number).

```
In [6]: area = pd.Series({'Alaska': 1723337, 'Texas': 695662,
                          'California': 423967}, name='area')
        population = pd.Series({'California': 39538223, 'Texas': 29145505,
                                'Florida': 21538187}, name='population')
```

```
In [7]: population / area
Out[7]: Alaska              NaN
        California    93.257784
        Florida             NaN
        Texas         41.896072
        dtype: float64
```

```
In [11]: A = pd.DataFrame(rng.integers(0, 20, (2, 2)),
                          columns=['a', 'b'])
         A
Out[11]:     a  b
         0  10  2
         1  16  9
```

```
In [12]: B = pd.DataFrame(rng.integers(0, 10, (3, 3)),
                          columns=['b', 'a', 'c'])
         B
Out[12]:    b  a  c
         0  5  3  1
         1  9  7  6
         2  4  8  5
```

```
In [13]: A + B
Out[12]:       a     b   c
         0  13.0   7.0 NaN
         1  23.0  18.0 NaN
         2   NaN   NaN NaN
```

Mechanisms for Missing

- Missing data is everywhere.
  - Survey data
  - Longitudinal Studies and Clinical trials
  - Recommendation systems
  - Data integration
- There are 3 major types of missing
  - Missing Completely at Random (MCAR):
    Pattern of missing indep of missing values and the values of any measured variables.
  - Missing at Random (MAR):
    Conditional on observed variables, missing indep of missing value.
  - Missing Not at Random (MNAR):
    Pattern of missing related to missing value, even after correcting for measured variables.

Dealing with Missing - 0

- Categorical case: Treat missing as an additional category.
- *Surrogate* variables: When a sample has a missing for a predictor in the tree, the surrogate predictors are then used to direct the sample toward the appropriate terminal node.
- Partial deletion:
  - *Listwise* deletion: An entire record is excluded from analysis if any single value is missing.
  - *Pairwise* deletion: Use the available data for each part of an analysis.

id	gender	age	result
1	Male	20	Positive
~~2~~	~~Female~~		~~Negative~~
3	Female	30	Positive
~~4~~		~~28~~	~~Negative~~
~~5~~	~~Female~~		~~Positive~~
6	Male	25	Positive
7	Male	21	Positive

Listwise deletion
(Complete case analysis)

id	gender	age	result
1	Male	20	Positive
2	Female		Negative
3	Female	30	Positive
4		28	Negative
5	Female		Positive
6	Male	25	Positive
7	Male	21	Positive

Pairwise deletion
(Available case analysis)

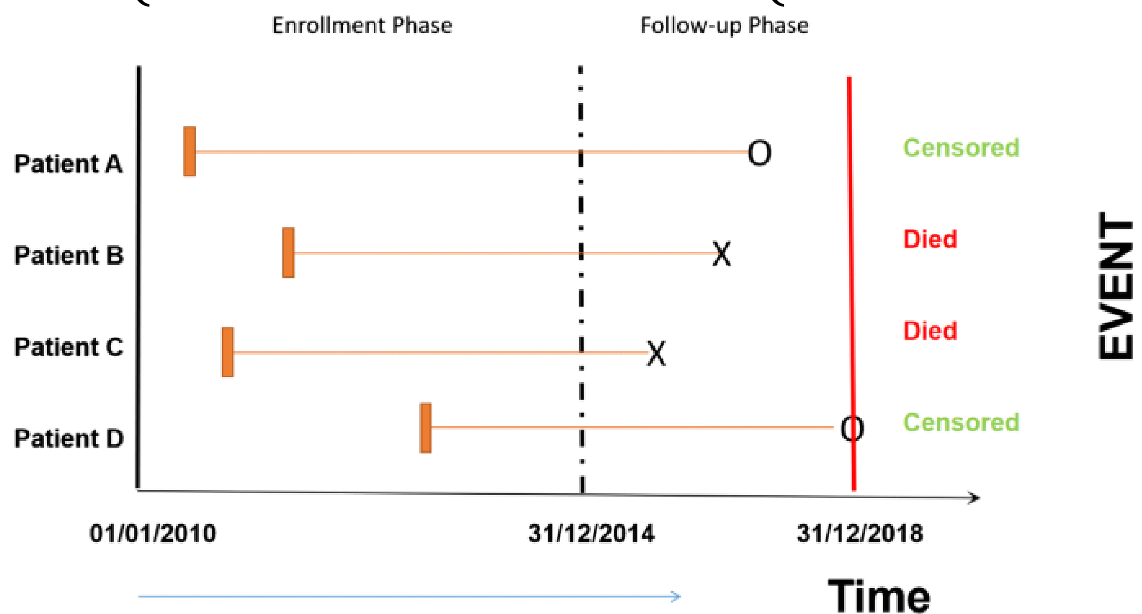
Dealing with Missing - 1

- Dummy variable adjustment: Add another variable in the database to indicate whether a value is missing.

$$X^* = \begin{cases} C, & D = 0 \\ X, & D = 1 \end{cases}$$

- Ex. Censored Data. The Full length of time is not observable, only observe right-censored data $(U_1, \delta_1), \dots, (U_n, \delta_n)$.

$$U_i = \begin{cases} T_i, & T_i \le C \\ C, & T_i > C \end{cases}, \qquad \delta_i = \begin{cases} 1, & T_i \le C \\ 0, & T_i > C \end{cases}$$



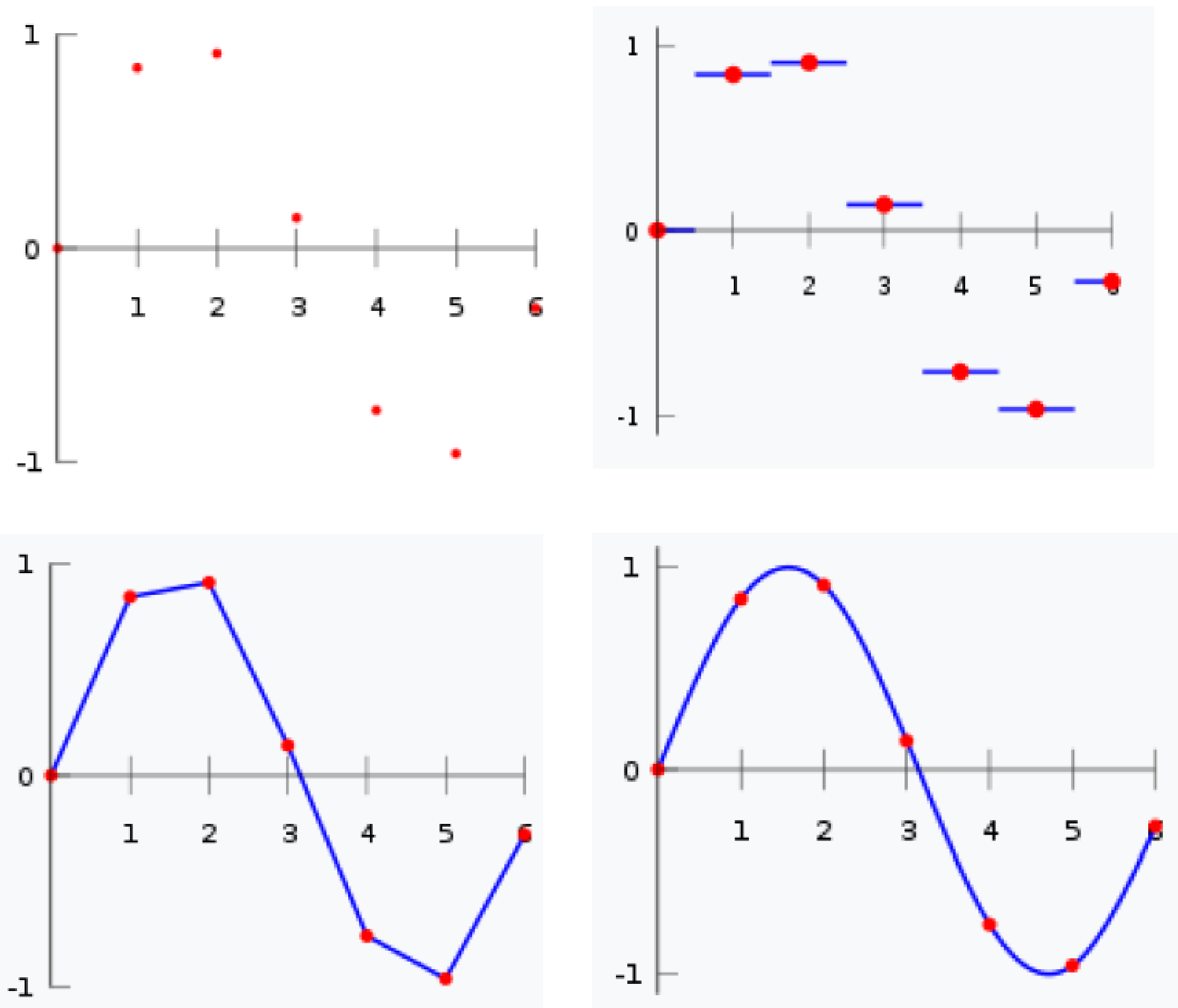
Dealing with Missing - 2

- *Imputation* Method: Replacing missing value with estimated values based on the available data. This can be done using various statistical methods.
  - Statistical method imputation
  - Constant imputation
  - Linear regression imputation
  - K-nearest neighbors imputation

- *Interpolation* Method: Estimating missing values by fitting a smooth curve that passes through the available data points.
  - Piecewise constant interpolation
  - Linear interpolation
  - Polynomial interpolation
  - Spline interpolation
  - Nearest-neighbor interpolation

Interpolation

x	$f(x)$
0	0
1	0.8415
2	0.9093
3	0.1411
4	−0.7568
5	−0.9589
6	−0.2794



Handling Missing - 0

- Generally, there are two strategies to track the presence of missing data in a table.
  - Using *mask* that globally indicates missing values.
  - Choosing a *sentinel value* that indicates a missing entry.
- For some data types, Pandas uses **None** as a sentinel value. **None** is a Python object, which means that any array containing **None** must have dtype = object.

```
In [2]: vals1 = np.array([1, None, 2, 3])
        vals1
Out[2]: array([1, None, 2, 3], dtype=object)
```

- The downside of using **None** in this way is that 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.

```
In [3]: %timeit np.arange(1E6, dtype=int).sum()
Out[3]: 2.73 ms ± 288 µs per loop (mean ± std. dev. of 7 runs, 100 loops each)

In [4]: %timeit np.arange(1E6, dtype=object).sum()
Out[4]: 92.1 ms ± 3.42 ms per loop (mean ± std. dev. of 7 runs, 10 loops each)
```

Handling Missing - 1

- The other missing data sentinel, **NaN.** It is a special floating-point value recognized by all systems that use the standard IEEE floating-point representation.

```
In [6]: vals2 = np.array([1, np.nan, 3, 4])
        vals2
Out[6]: array([ 1., nan,  3.,  4.])
```

- Keep in mind that, **NaN** is a bit like a data virus. It infects any other object it touches.

```
In [7]: 1 + np.nan
Out[7]: nan

In [8]: 0 * np.nan
Out[8]: nan

In [9]: vals2.sum(), vals2.min(), vals2.max()
Out[9]: (nan, nan, nan)

In [10]: np.nansum(vals2), np.nanmin(vals2), np.nanmax(vals2)
Out[10]: (8.0, 1.0, 4.0)
```

Handling Missing - 2

- **NaN** and **None** both have their place. Pandas is built to handle the two of them nearly interchangeably, converting between then where appropriate.

```
In [11]: pd.Series([1, np.nan, 2, None])
Out[11]: 0    1.0
         1    NaN
         2    2.0
         3    NaN
         dtype: float64
```

- For types that don’t have an available sentinel value, Pandas automatically typecasts when **NaN** values are present.

```
In [12]: x = pd.Series(range(2), dtype=int)
         x
Out[12]: 0    0
         1    1
         dtype: int64
```

```
In [13]: x[0] = None
         x
Out[13]: 0    NaN
         1    1.0
         dtype: float64
```

- Keep in mind that in Pandas, string data is always stored with an *object* dtype.

國立臺北大學
National Taipei University

Nullable Dtype

- In early versions of Pandas, **NaN** and **None** as sentinel values were the only missing data representations available. The primary difficulty this introduced was with regard to the implicit type casting.

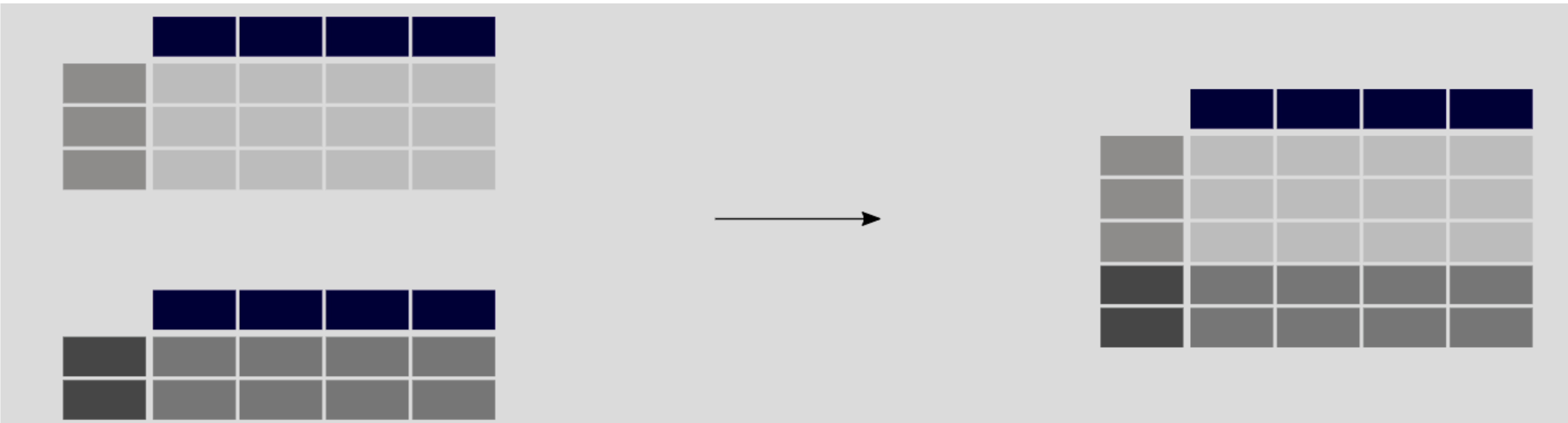
- To address this difficulty, Pandas later added *nullable dtypes*, which are distinguished from regular dtypes by capitalization of their names. For backward compatibility, these *nullable dtypes* are only used if specifically requested.

```
In [14]: pd.Series([1, np.nan, 2, None, pd.NA], dtype='Int32')
Out[14]: 0       1
         1    <NA>
         2       2
         3    <NA>
         4    <NA>
         dtype: Int32
```

- Pandas treats **None**, **NaN**, and **NA** as essentially interchangeable for indicating missing or null values.

Combine data from Multiple Tables

- The pd.concat function performs concatenation operations of multiple tables along one of the axes (row-wise or column-wise)

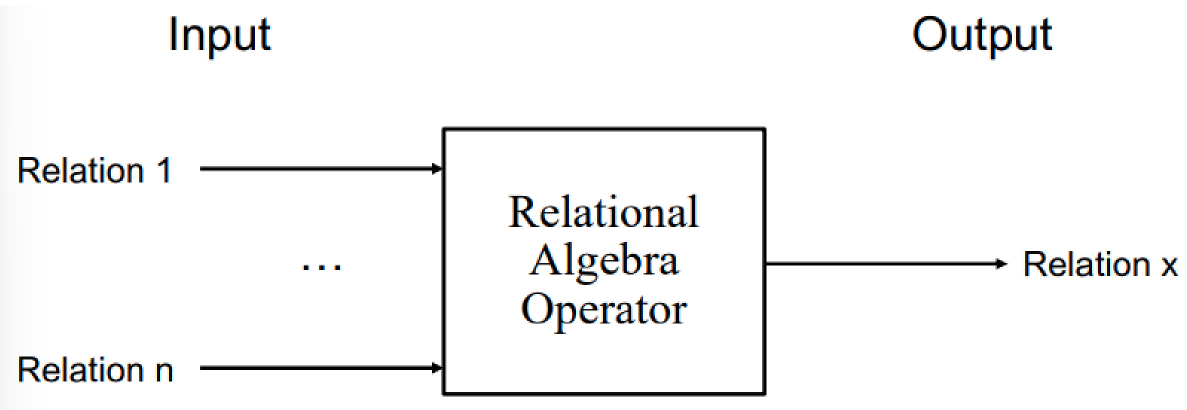


- One important difference between np.concatenate and pd.concat is that Pandas concatenation preserves indices, even if the result will have duplicate indices.

```
Out[9]: x              y              pd.concat([x, y])
          A   B             A   B             A   B
      0   A0  B0        0   A2  B2        0   A0  B0
      1   A1  B1        1   A3  B3        1   A1  B1
                                          0   A2  B2
                                          1   A3  B3
```

Merge and Join

- The behavior implemented in pd.merge is a subset of what is known as *relational algebra*.
- Six basis operators in *relational algebra*:
  - Select (σ): used to filter rows based on a certain condition.
  - Project (π): which selects only the columns specified.
  - Cross product ($\times$): allows to combine 2 relations.
  - Set difference ($-$): returns every row in the 1st table except the rows that also show up in the 2nd table.
  - Union ($\cup$): take all the rows from each tuple and combine them removing duplicates along the way.
  - Rename (ρ): renames attributes and relations.



Projection (π)

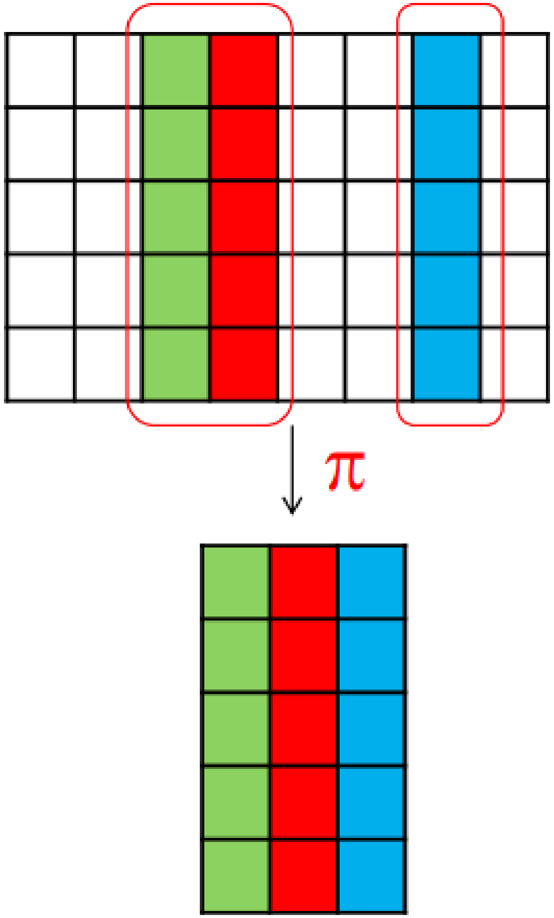
- Notation:

$$\pi_{A_1,\ldots,A_n}(R)$$

where $A_1, \cdots, A_n$ are attribute names and R is relation.

- Ex.

$$\pi_{\mathrm{A,C}}(\mathrm{r})$$



r

A	B	C
α	10	2
α	20	2
β	30	2
β	40	4

$\pi_{A,C}(r)$

A	C
α	2
β	2
β	4

Selection (σ)

- Notation:

$$\sigma_P(R)$$

where R is relation and P is condition.

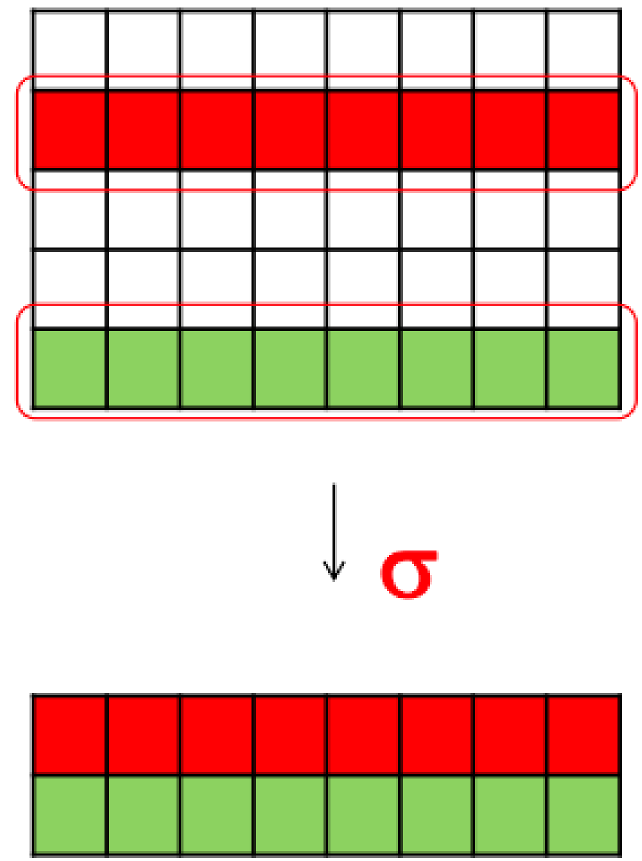
- Ex.

$$\sigma_{\text{A=B and } D>5}(r)$$

A	B	C	D
α	α	1	7
α	β	5	7
β	β	12	3
β	β	23	10

$\sigma_{A=B \wedge D>5}(r)$

A	B	C	D
α	α	1	7
β	β	23	10



Cross Product (×)

- Notation:

$$\mathrm{R} \times S = \{tq \mid t \in R \text{ and } q \in S\}$$

- Ex.

$$r \times s$$

r

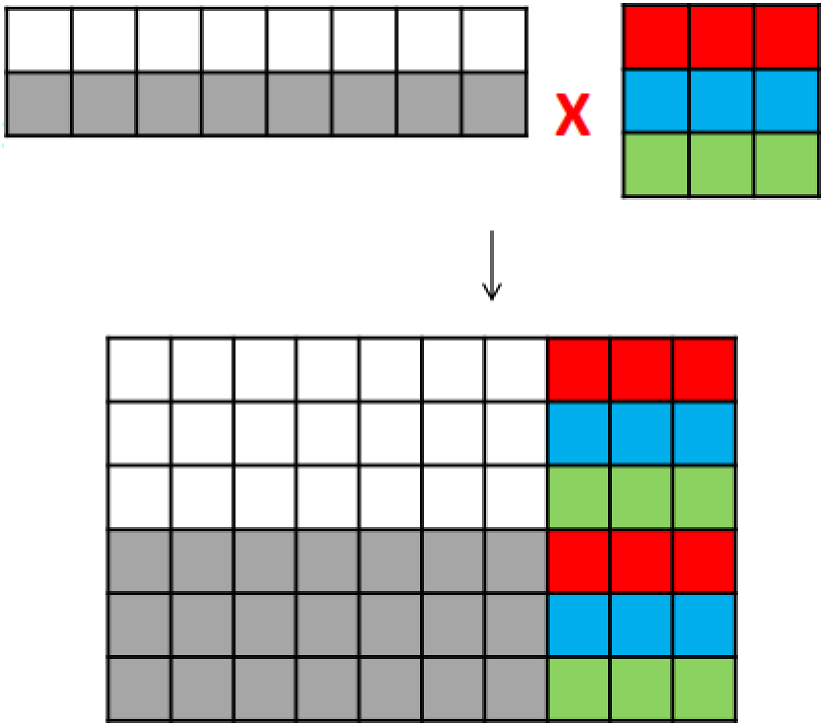
A	B
α	1
β	2

s

C	D	E
α	10	+
β	10	+
β	20	−
γ	10	−

$r \times s$

A	B	C	D	E
α	1	α	10	+
α	1	β	10	+
α	1	β	20	−
α	1	γ	10	−
β	2	α	10	+
β	2	β	10	+
β	2	β	20	−
β	2	γ	10	−



Union (∪)

- Notation:

$$R \cup S = \{t \mid t \in R \text{ or } t \in S\}$$

- Ex.

$$r \cup s$$

r

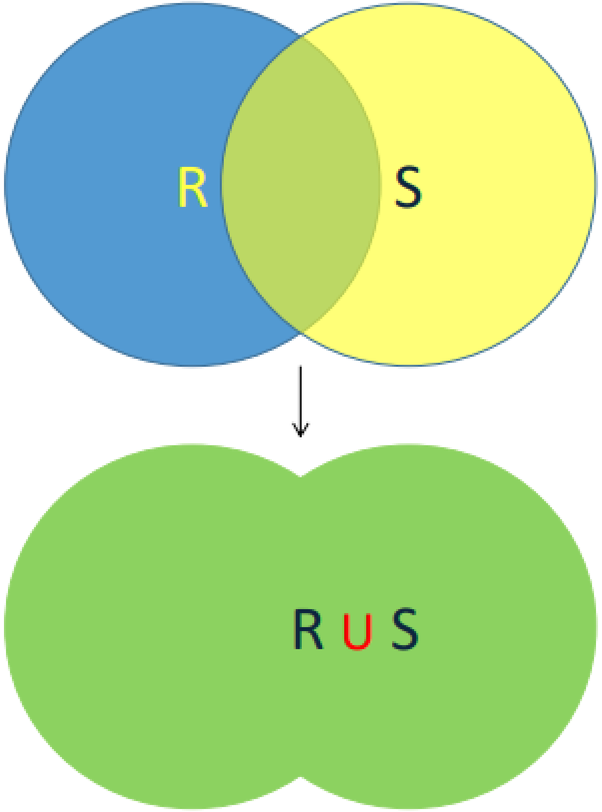
A	B
α	1
α	2
β	1

s

A	B
α	2
β	3

$r \cup s$

A	B
α	1
α	2
β	1
β	3



Set Difference ($-$)

- Notation:

$$R - S = \{t | t \in R \text{ and } t \notin S\}$$

- Ex.

$$r - s$$

r

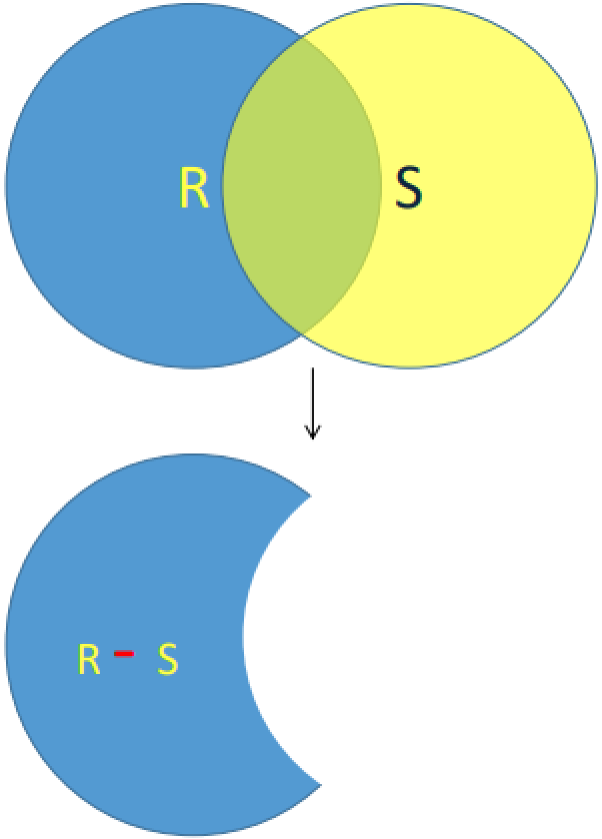
A	B
α	1
α	2
β	1

s

A	B
α	2
β	3

$r - s$

A	B
α	1
β	1



Rename (ρ)

- Notation:
  - Rename relation

$$\rho_S(R)$$

  - Rename attributes

$$\rho_{(B_1,\cdots,B_n)}(R)$$

  - Rename relation and its attributes

$$\rho_{S(B_1,\cdots,B_n)}(R)$$

  - If not all attributes are renamed, can specify renamed attributes

$$\rho_{S(a\to a^*,b\to b^*)}(R)$$

- Ex.

$$\rho_{\mathrm{C4Profs}}(\sigma_{\mathrm{Level=C4}}(\mathrm{Professors}))$$

Input

Professors		
PersNr	**Name**	**Level**
2125	Sokrates	C4
2126	Russel	C4
2127	Kopernikus	C3

$\xrightarrow{\sigma_{\text{Level= 'C4'}}}$

Input

Professors		
PersNr	**Name**	**Level**
2125	Sokrates	C4
2126	Russel	C4

$\xrightarrow{\rho_{\text{C4Profs}}}$

Output

C4Profs		
PersNr	**Name**	**Level**
2125	Sokrates	C4
2126	Russel	C4

Categories of Joins - 0

- The *pd.merge* function implements a number of types of joins:
  - One-to-One Joins: Each row in one table is linked (or related) to a single row in another table using a *key* column

```
In [2]: df1 = pd.DataFrame({'employee': ['Bob', 'Jake', 'Lisa', 'Sue'],
                            'group': ['Accounting', 'Engineering',
                                      'Engineering', 'HR']})
        df2 = pd.DataFrame({'employee': ['Lisa', 'Bob', 'Jake', 'Sue'],
                            'hire_date': [2004, 2008, 2012, 2014]})
        display('df1', 'df2')
Out[2]: df1                                 df2
          employee        group               employee  hire_date
        0      Bob   Accounting             0     Lisa       2004
        1     Jake  Engineering             1      Bob       2008
        2     Lisa  Engineering             2     Jake       2012
        3      Sue           HR             3      Sue       2014
```

```
In [3]: df3 = pd.merge(df1, df2)
        df3
Out[3]:   employee        group  hire_date
        0      Bob   Accounting       2008
        1     Jake  Engineering       2012
        2     Lisa  Engineering       2004
        3      Sue           HR       2014
```

Categories of Joins - 1

- Many-to-One Joins: Each row in one table is linked (or related) to one, or more, rows in another table using a *key* column.

```
In [4]: df4 = pd.DataFrame({'group': ['Accounting', 'Engineering', 'HR'],
                            'supervisor': ['Carly', 'Guido', 'Steve']})
        display('df3', 'df4', 'pd.merge(df3, df4)')
Out[4]: df3                                     df4
          employee        group  hire_date              group supervisor
        0      Bob   Accounting       2008      0   Accounting      Carly
        1     Jake  Engineering       2012      1  Engineering      Guido
        2     Lisa  Engineering       2004      2           HR      Steve
        3      Sue           HR       2014

        pd.merge(df3, df4)
          employee        group  hire_date supervisor
        0      Bob   Accounting       2008      Carly
        1     Jake  Engineering       2012      Guido
        2     Lisa  Engineering       2004      Guido
        3      Sue           HR       2014      Steve
```

Categories of Joins - 2

- Many-to-Many Joins: One, or more, rows in one table is linked (or related) to one, or more, rows in another table using a *key* column.

```
In [5]: df5 = pd.DataFrame({'group': ['Accounting', 'Accounting',
                                      'Engineering', 'Engineering', 'HR', 'HR'],
                            'skills': ['math', 'spreadsheets', 'software', 'math',
                                       'spreadsheets', 'organization']})
        display('df1', 'df5', "pd.merge(df1, df5)")
Out[5]: df1                          df5
```

df1

	employee	group
0	Bob	Accounting
1	Jake	Engineering
2	Lisa	Engineering
3	Sue	HR

df5

	group	skills
0	Accounting	math
1	Accounting	spreadsheets
2	Engineering	software
3	Engineering	math
4	HR	spreadsheets
5	HR	organization

pd.merge(df1, df5)

	employee	group	skills
0	Bob	Accounting	math
1	Bob	Accounting	spreadsheets
2	Jake	Engineering	software
3	Jake	Engineering	math
4	Lisa	Engineering	software
5	Lisa	Engineering	math
6	Sue	HR	spreadsheets
7	Sue	HR	organization

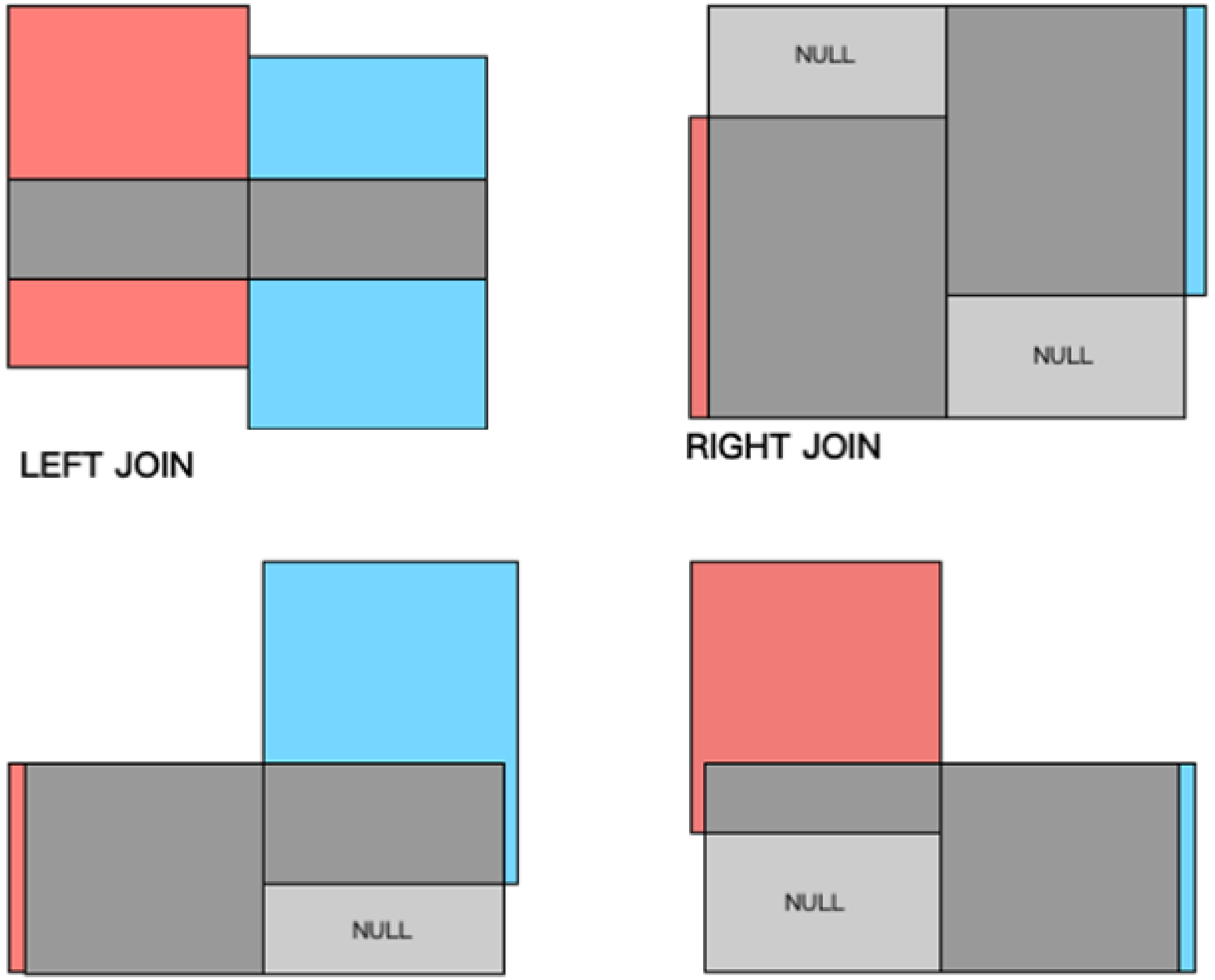
Set Arithmetic for Joins

JOIN（INNER JOIN）

FULL JOIN

LEFT JOIN

RIGHT JOIN



Simple Aggregation in Pandas

- As with a 1-dim NumPy array, for a Pandas *Series* the aggregates return a single value.

```
In [4]: rng = np.random.RandomState(42)
        ser = pd.Series(rng.rand(5))
        ser
Out[4]: 0    0.374540
        1    0.950714
        2    0.731994
        3    0.598658
        4    0.156019
        dtype: float64
```

```
In [5]: ser.sum()
Out[5]: 2.811925491708157

In [6]: ser.mean()
Out[6]: 0.5623850983416314
```

- For a *DataFrame*, by default the aggregates return results within each column.

```
In [7]: df = pd.DataFrame({'A': rng.rand(5),
                           'B': rng.rand(5)})
        df
Out[7]:           A         B
        0  0.155995  0.020584
        1  0.058084  0.969910
        2  0.866176  0.832443
        3  0.601115  0.212339
        4  0.708073  0.181825

In [8]: df.mean()
Out[8]: A    0.477888
        B    0.443420
        dtype: float64
```

Aggregation	Returns
`count`	Total number of items
`first`, `last`	First and last item
`mean`, `median`	Mean and median
`min`, `max`	Minimum and maximum
`std`, `var`	Standard deviation and variance
`mad`	Mean absolute deviation
`prod`	Product of all items
`sum`	Sum of all items

Group by + Aggregation

- Split, Apply, Combine operation, where apply is a summation aggregation.

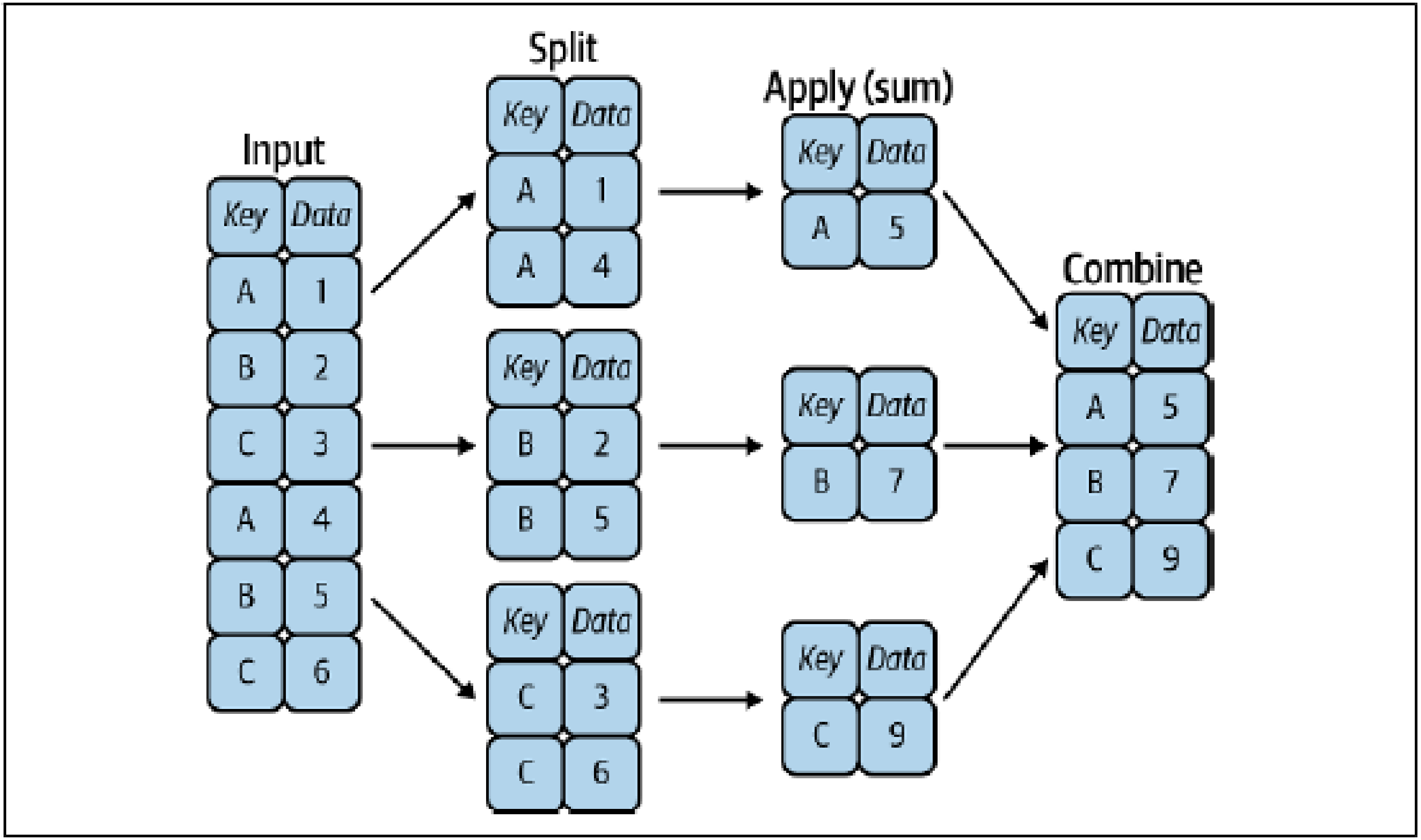


Figure 20-1. A visual representation of a groupby operation[1]

apply vs transform

- Input
  - *apply*: implicitly passes all the columns for each group as a *DataFrame* to the custom function.
  - *transform*: passes each column for each group individually as a *Series* to the custom function.

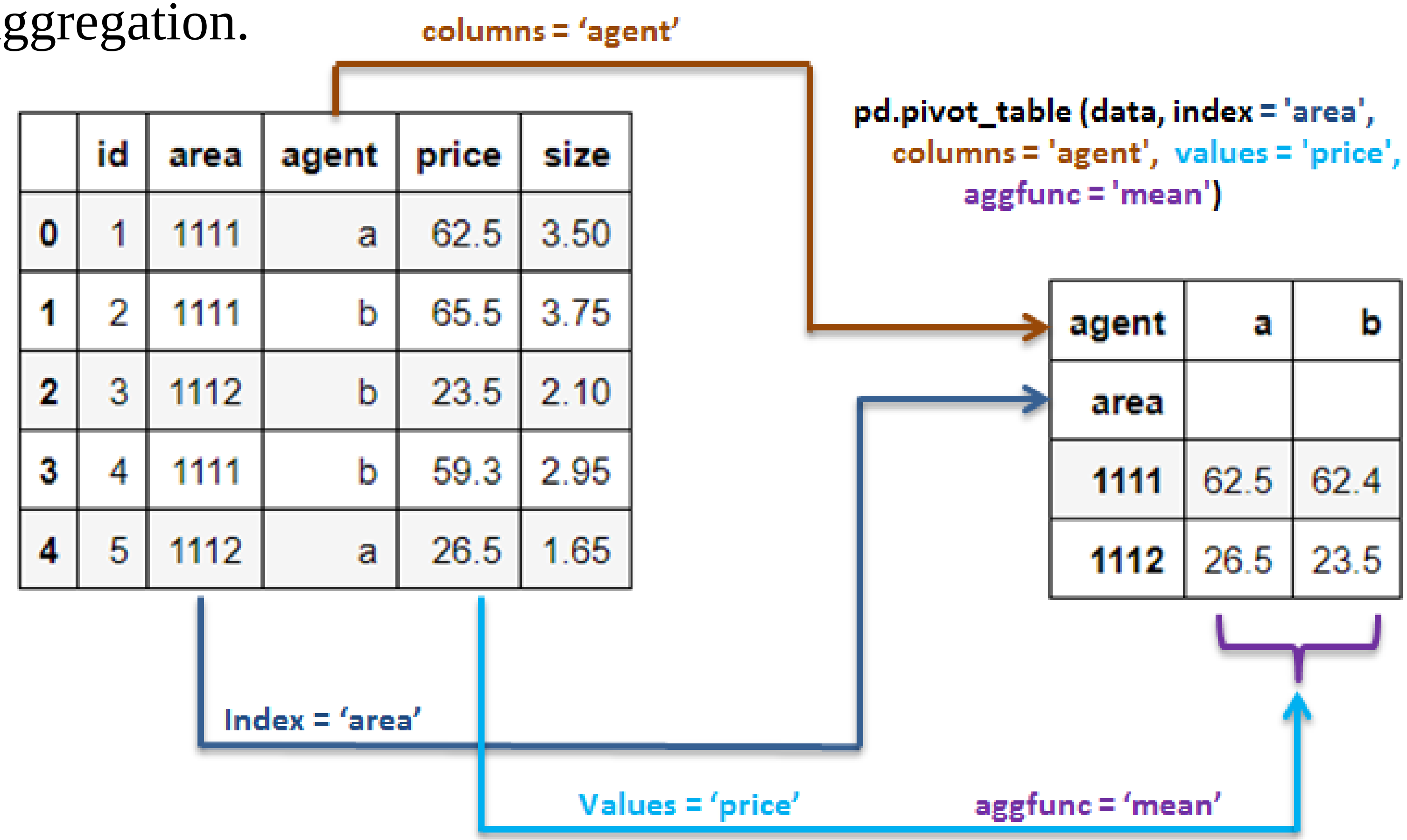
- Output
  - The custom function passed to *apply* can return a scalar, or a *Series* or *DataFrame* (or NumPy array or even list).
  - The custom function passed to transform must return a seq (1-dim Series, array or list) the same length as the group.

- *transform* works on just one *Series* at a time

  apply works on the entire *DataFrame* at once.

Pivot

- The pivot table takes simple column-wise data as input, and groups the entries into a two-dimensional table that provides a multi-dim summarization of the data.
- Think of pivot tables as essentially a multi-dim version of groupby aggregation.



	id	area	agent	price	size
0	1	1111	a	62.5	3.50
1	2	1111	b	65.5	3.75
2	3	1112	b	23.5	2.10
3	4	1111	b	59.3	2.95
4	5	1112	a	26.5	1.65

agent	a	b
area		
1111	62.5	62.4
1112	26.5	23.5