5QQMN534: Algorithmic Finance

Week3: Mastering Basics – Financial Data Analysis with Pandas

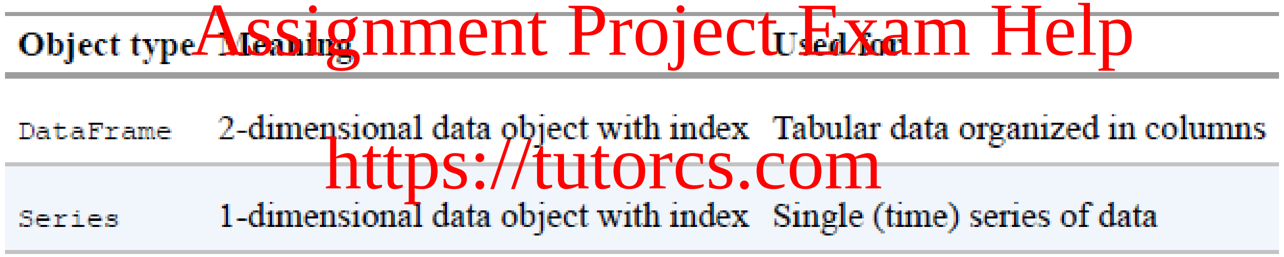
Yves Hilpisch - Python for Finance 2nd Edition 2019: Chapter 5

Agenda

- Data Analysis with Pandas
- The DataFrame Class
  - First Steps with a DataFrame Class
  - * .loc and .iloc functions
  - Second Steps with a DataFrame Class
- Basic Analytics
- Basic Visualisation
- The Series Class
- GroupBy Operations
- Complex Selection
- Concatenation, Joining and Merging
- Performance Aspects
- Conclusion

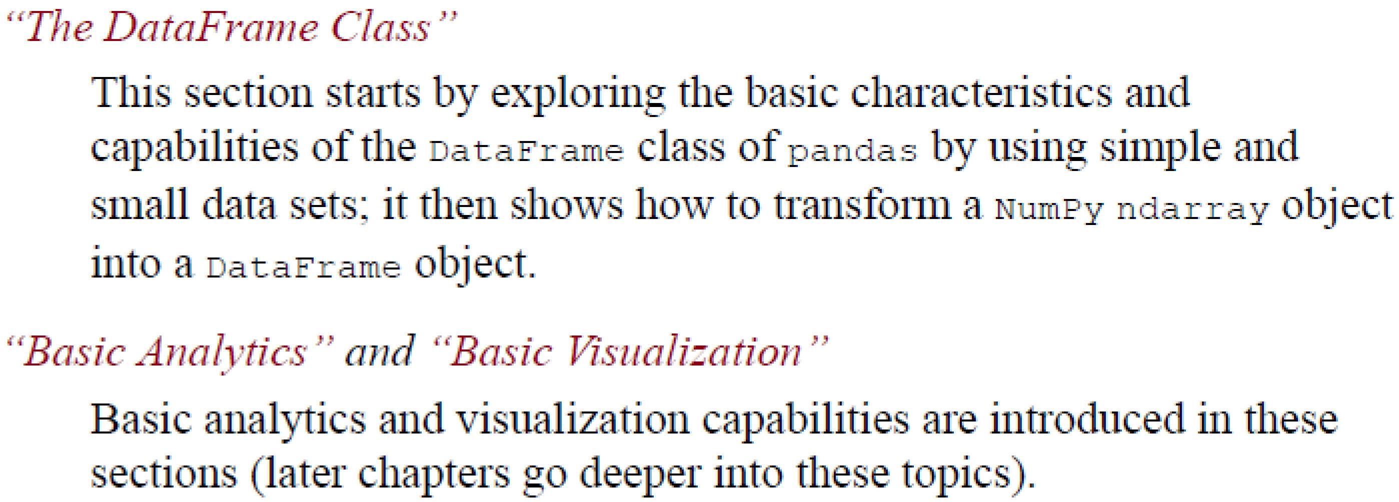
Data Analysis with pandas1

- This chapter is about `pandas`, a library for data analysis with a focus on tabular data. `pandas` is a powerful tool that not only provides many useful classes and functions but also does a great job of wrapping functionality from other packages.
- The result is a user interface that makes data analysis, and in particular financial analysis, a convenient and efficient task. This chapter covers the following fundamental data structures:



Object type	Meaning	Used for
`DataFrame`	2-dimensional data object with index	Tabular data organized in columns
`Series`	1-dimensional data object with index	Single (time) series of data

The chapter is organized as follows:



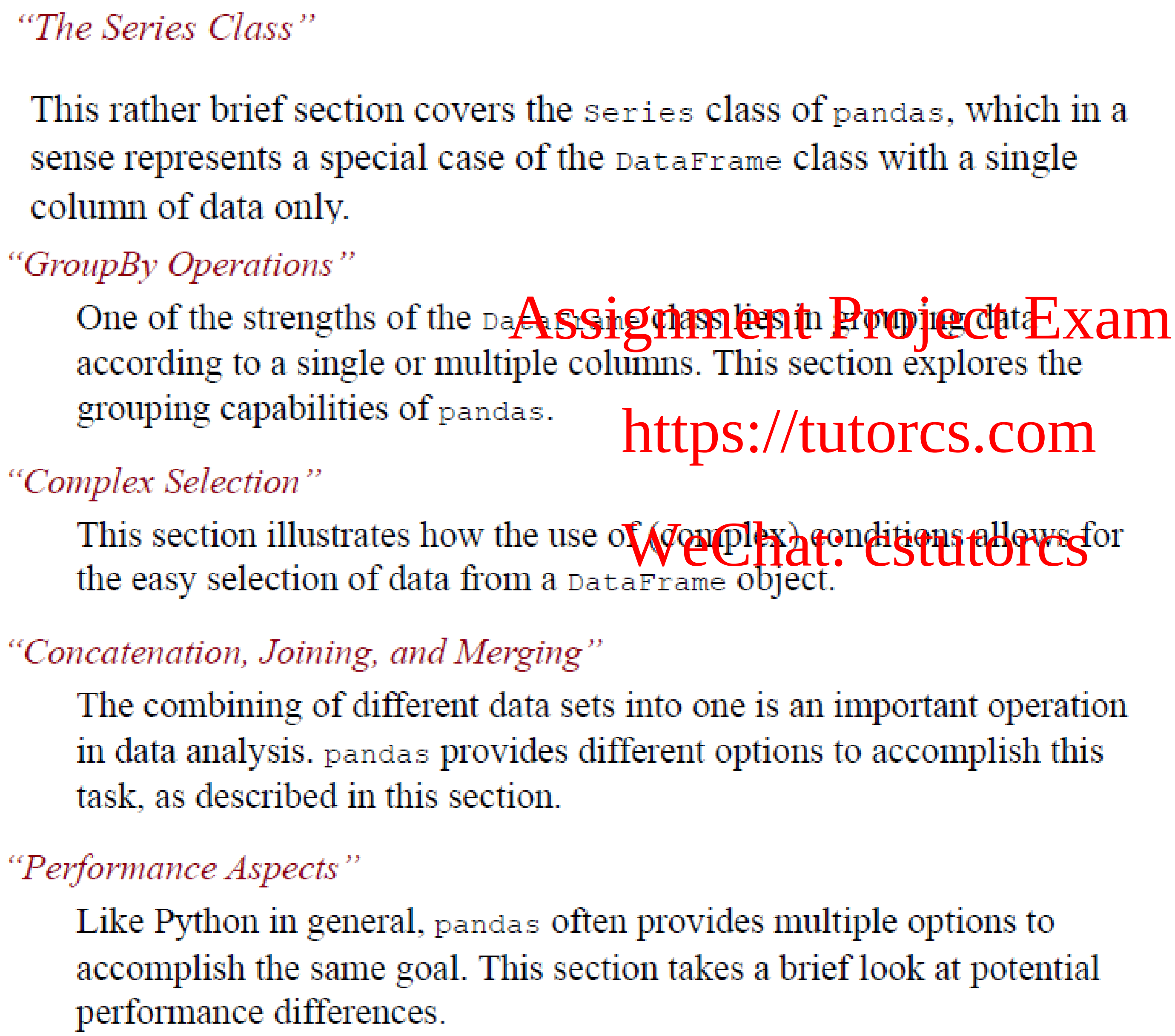
"The DataFrame Class"

This section starts by exploring the basic characteristics and capabilities of the `DataFrame` class of `pandas` by using simple and small data sets; it then shows how to transform a `NumPy` `ndarray` object into a `DataFrame` object.

"Basic Analytics" and "Basic Visualization"

Basic analytics and visualization capabilities are introduced in these sections (later chapters go deeper into these topics).

Data Analysis with pandas2



"The Series Class"

This rather brief section covers the `Series` class of `pandas`, which in a sense represents a special case of the `DataFrame` class with a single column of data only.

"GroupBy Operations"

One of the strengths of the `DataFrame` class lies in grouping data according to a single or multiple columns. This section explores the grouping capabilities of `pandas`.

"Complex Selection"

This section illustrates how the use of (complex) conditions allows for the easy selection of data from a `DataFrame` object.

"Concatenation, Joining, and Merging"

The combining of different data sets into one is an important operation in data analysis. `pandas` provides different options to accomplish this task, as described in this section.

"Performance Aspects"

Like Python in general, `pandas` often provides multiple options to accomplish the same goal. This section takes a brief look at potential performance differences.

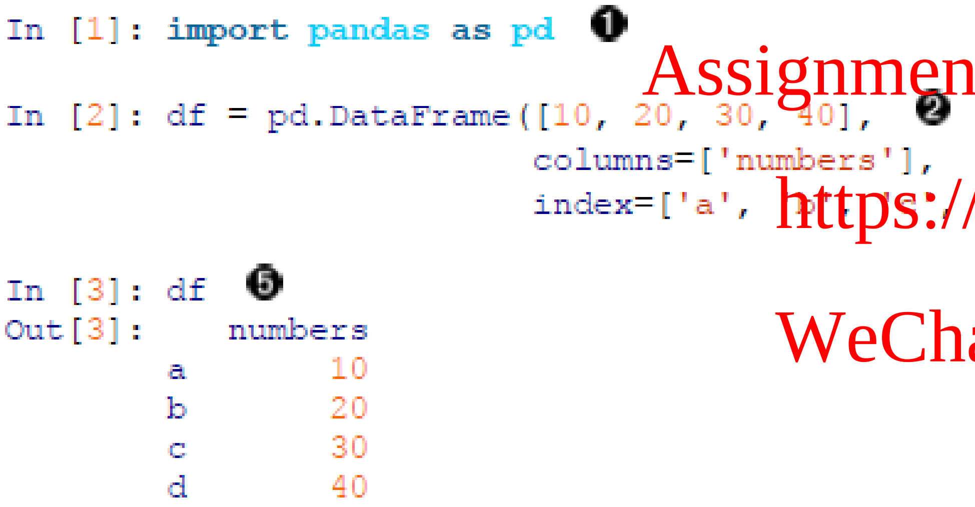
The DataFrame Class

- At the core of `pandas` (and this chapter) is the `DataFrame`, a class designed to efficiently handle data in tabular form — i.e., data characterized by a columnar organization.
- To this end, the `DataFrame` class provides, for instance, column labelling as well as flexible indexing capabilities for the rows (records) of the data set, similar to a table in a relational database or an Excel spreadsheet.
- This section covers some fundamental aspects of the `pandas DataFrame` class.
- The class is so complex and powerful that only a fraction of its capabilities can be presented here.
- Subsequent chapters provide more examples and shed light on different aspects.

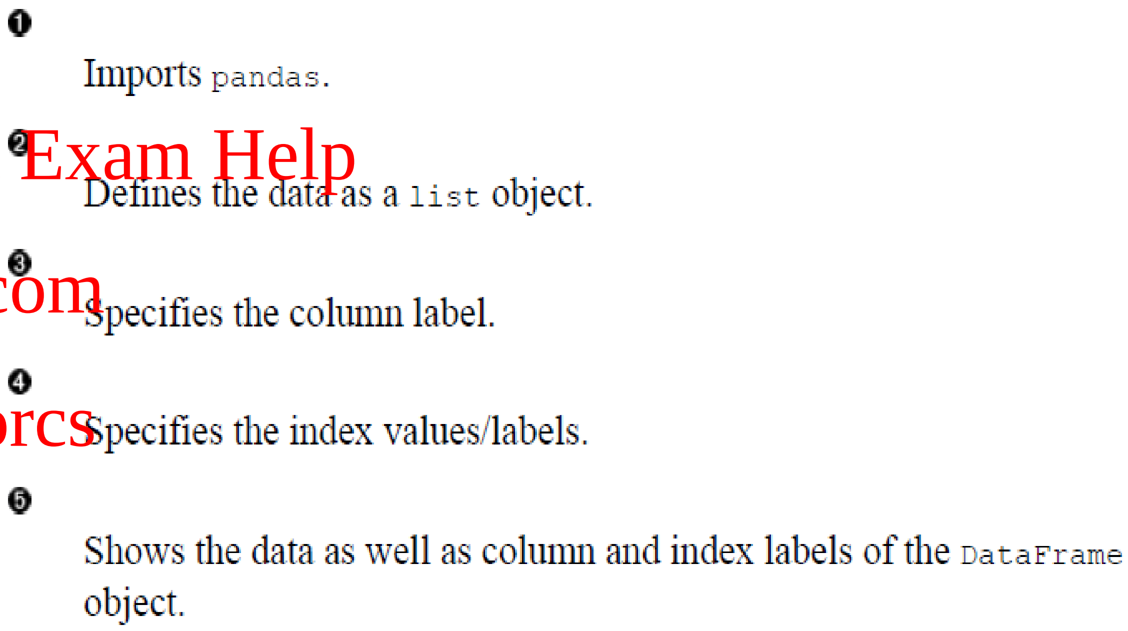
Documentation: https://pandas.pydata.org/pandas-docs/stable/reference/frame.html

First Steps with a DataFrame Class1

- On a fundamental level, the `DataFrame` class is designed to manage indexed and labelled data, not too different from a SQL database table or a worksheet in a spreadsheet application.
- Consider the following creation of a `DataFrame` object



```
In [1]: import pandas as pd  ❶

In [2]: df = pd.DataFrame([10, 20, 30, 40],  ❷
                          columns=['numbers'],  ❸
                          index=['a', 'b', 'c', 'd'])  ❹

In [3]: df  ❺
Out[3]:    numbers
        a       10
        b       20
        c       30
        d       40
```



❶
Imports `pandas`.

❷
Defines the data as a `list` object.

❸
Specifies the column label.

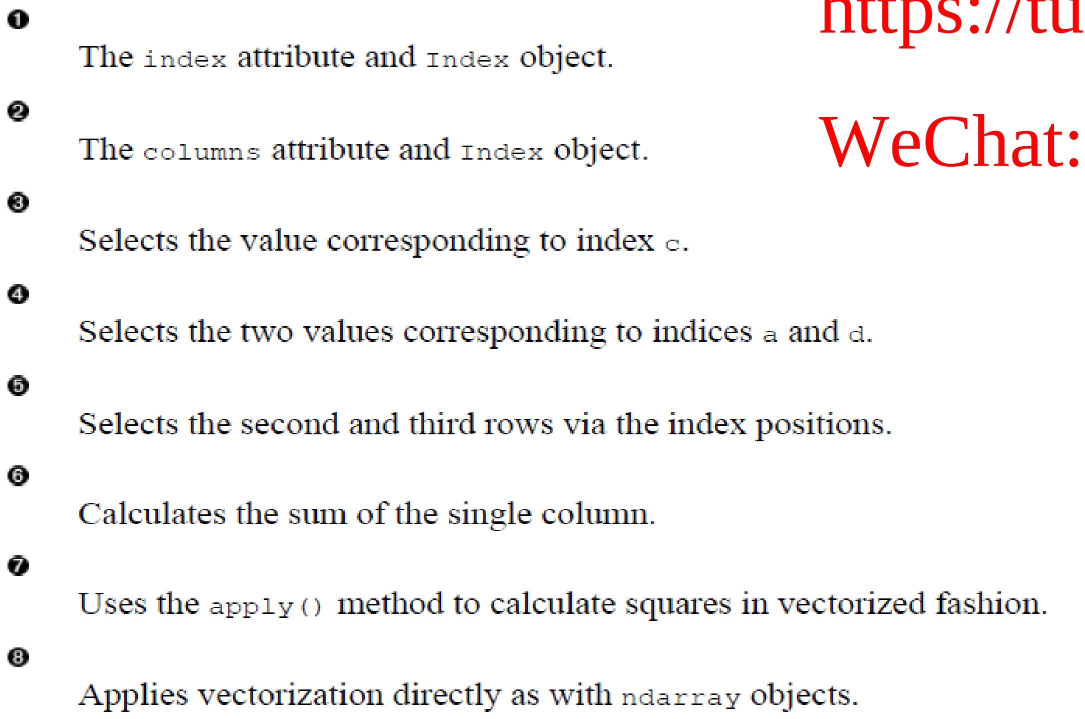
❹
Specifies the index values/labels.

❺
Shows the data as well as column and index labels of the `DataFrame` object.

- This simple example already shows some major features of the `DataFrame` class when it comes to storing data:
- Data itself can be provided in different shapes and types (`list`, `tuple`, `ndarray`, and `dict` objects are candidates).
- Data is organized in columns, which can have custom names (labels).
- There is an index that can take on different formats (e.g., numbers, strings, time information).

First Steps with a DataFrame Class2

- Working with a `DataFrame` object is in general pretty convenient and efficient with regard to the handling of the object, e.g., compared to regular `ndarray` objects, which are more specialized and more restricted when one wants to (say) enlarge an existing object.
- At the same time, `DataFrame` objects are often as computationally efficient as `ndarray` objects.
- The following are simple examples showing how typical operations on a `DataFrame` object work:



❶ The `index` attribute and `Index` object.

❷ The `columns` attribute and `Index` object.

❸ Selects the value corresponding to index `c`.

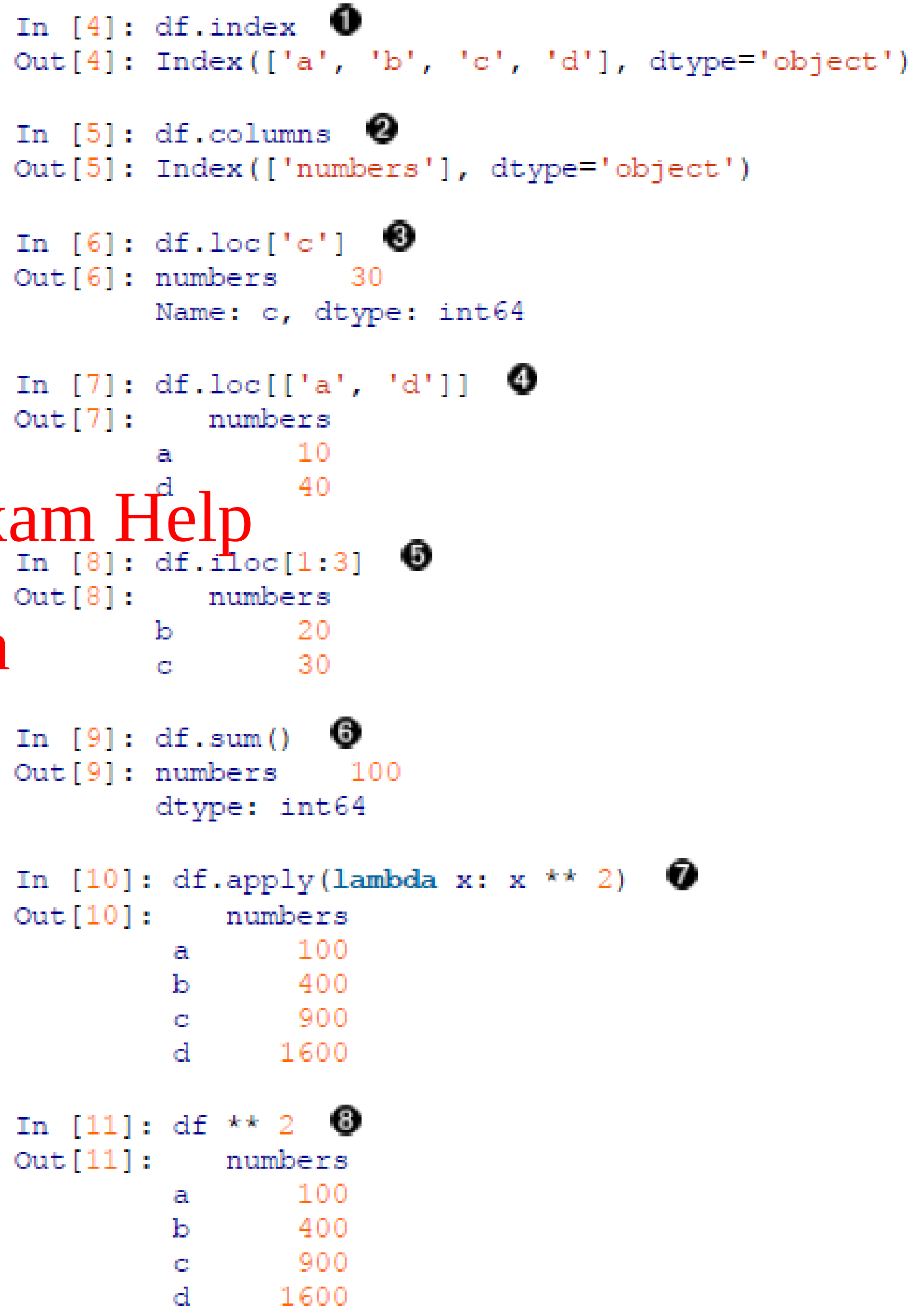
❹ Selects the two values corresponding to indices `a` and `d`.

❺ Selects the second and third rows via the index positions.

❻ Calculates the sum of the single column.

❼ Uses the `apply()` method to calculate squares in vectorized fashion.

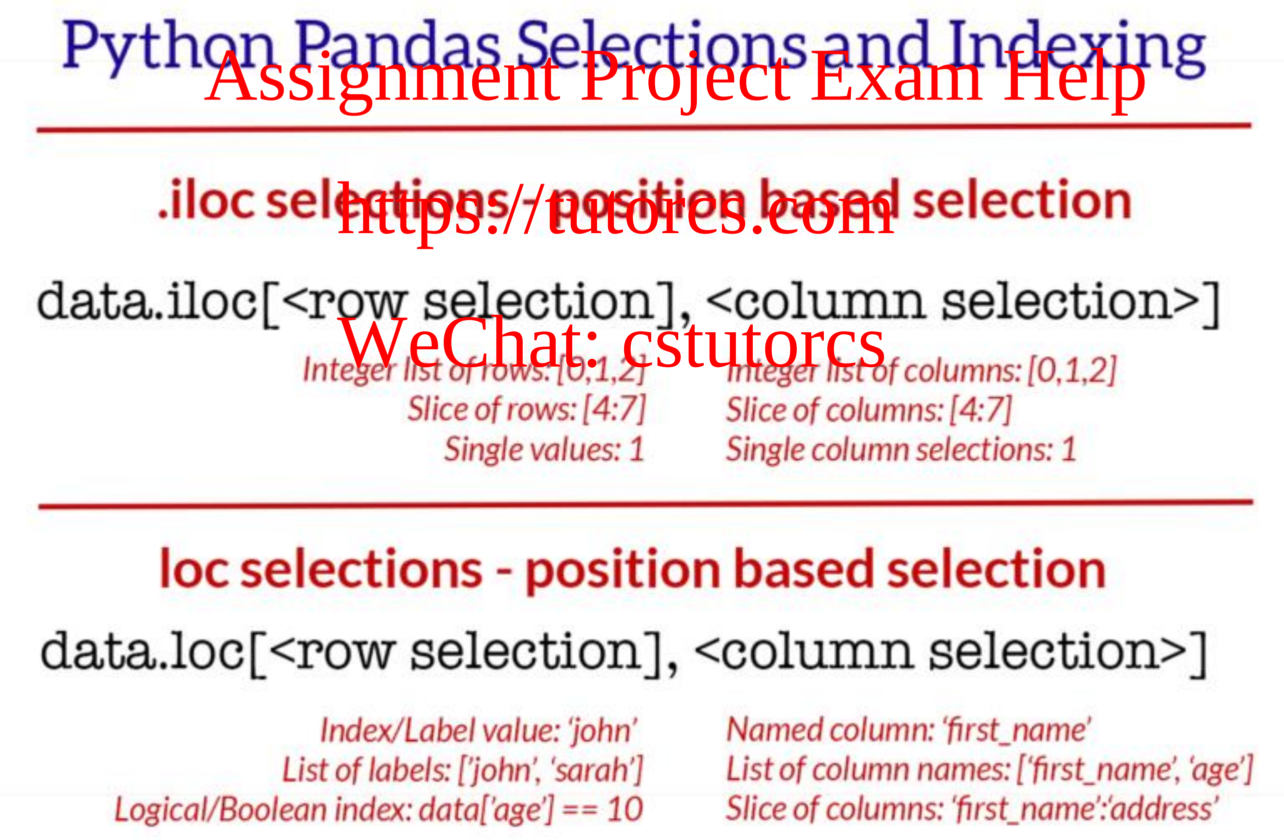
❽ Applies vectorization directly as with `ndarray` objects.



```
In [4]: df.index ❶
Out[4]: Index(['a', 'b', 'c', 'd'], dtype='object')

In [5]: df.columns ❷
Out[5]: Index(['numbers'], dtype='object')

In [6]: df.loc['c'] ❸
Out[6]: numbers    30
        Name: c, dtype: int64

In [7]: df.loc[['a', 'd']] ❹
Out[7]:    numbers
        a       10
        d       40

In [8]: df.iloc[1:3] ❺
Out[8]:    numbers
        b       20
        c       30

In [9]: df.sum() ❻
Out[9]: numbers    100
        dtype: int64

In [10]: df.apply(lambda x: x ** 2) ❼
Out[10]:    numbers
         a      100
         b      400
         c      900
         d     1600

In [11]: df ** 2 ❽
Out[11]:    numbers
         a      100
         b      400
         c      900
         d     1600
```

*.loc and .iloc functions

- loc gets rows (or columns) with **particular** labels from the index. Can use DateTime Indexes
- iloc gets rows (or columns) at particular positions in the index (so it **only** takes integers).



Python Pandas Selections and Indexing

.iloc selections - position based selection

data.iloc[<row selection>, <column selection>]

Integer list of rows: [0,1,2]
Slice of rows: [4:7]
Single values: 1

Integer list of columns: [0,1,2]
Slice of columns: [4:7]
Single column selections: 1

loc selections - position based selection

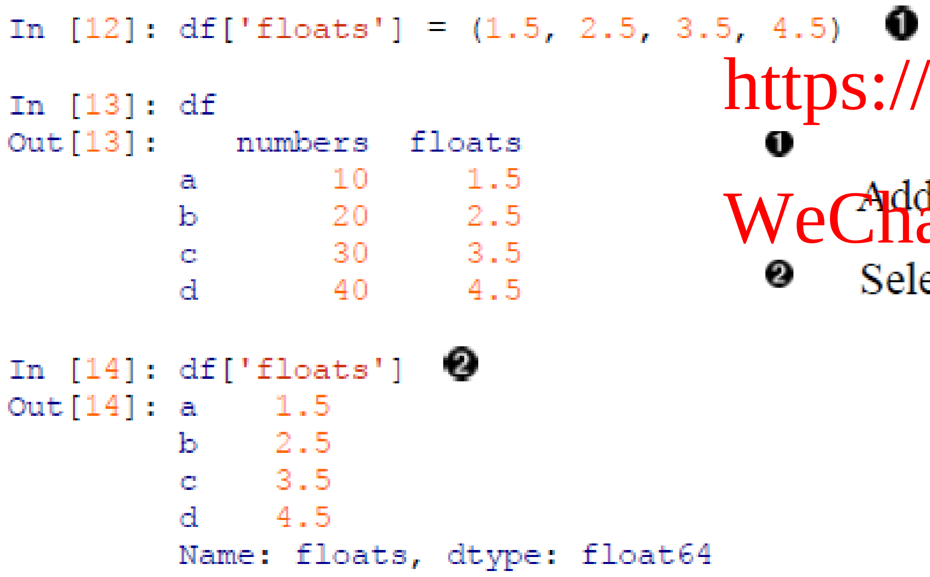
data.loc[<row selection>, <column selection>]

Index/Label value: 'john'
List of labels: ['john', 'sarah']
Logical/Boolean index: data['age'] == 10

Named column: 'first_name'
List of column names: ['first_name', 'age']
Slice of columns: 'first_name':'address'

First Steps with a DataFrame Class3

- Contrary to `NumPy ndarray` objects, enlarging the `DataFrame` object in both dimensions is possible:



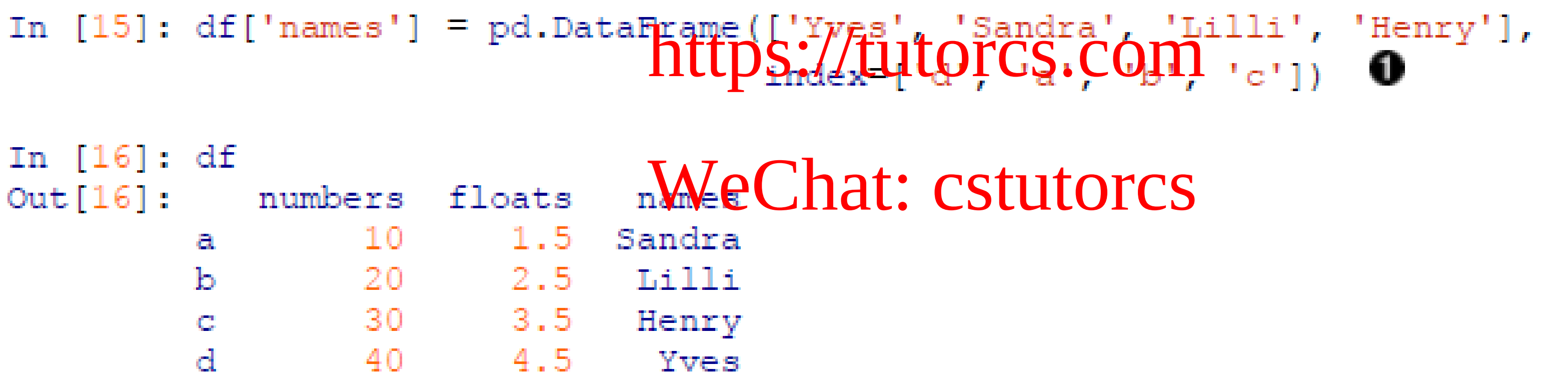
```
In [12]: df['floats'] = (1.5, 2.5, 3.5, 4.5)  ❶

In [13]: df
Out[13]:    numbers  floats
         a       10     1.5
         b       20     2.5
         c       30     3.5
         d       40     4.5

In [14]: df['floats']  ❷
Out[14]: a    1.5
         b    2.5
         c    3.5
         d    4.5
         Name: floats, dtype: float64
```

❶ Adds a new column with `float` objects provided as a `tuple` object.

❷ Selects this column and shows its data and index labels.

First Steps with a DataFrame Class4

- A whole `DataFrame` object can also be taken to define a new column.
- In such a case, indices are aligned automatically:



```
In [15]: df['names'] = pd.DataFrame(['Yves', 'Sandra', 'Lilli', 'Henry'],
                                    index=['d', 'a', 'b', 'c'])  ❶

In [16]: df
Out[16]:    numbers  floats   names
         a       10     1.5  Sandra
         b       20     2.5   Lilli
         c       30     3.5   Henry
         d       40     4.5    Yves
```

❶ Another new column is created based on a `DataFrame` object.

First Steps with a DataFrame Class5

- Appending data works similarly.
- However, in the following example a side effect is seen that is usually to be avoided - namely, the index gets replaced by a simple range index:

❶

Appends a new row via a `dict` object; this is a temporary operation during which index information gets lost.

❷

Appends the row based on a `DataFrame` object with index information; the original index information is preserved.

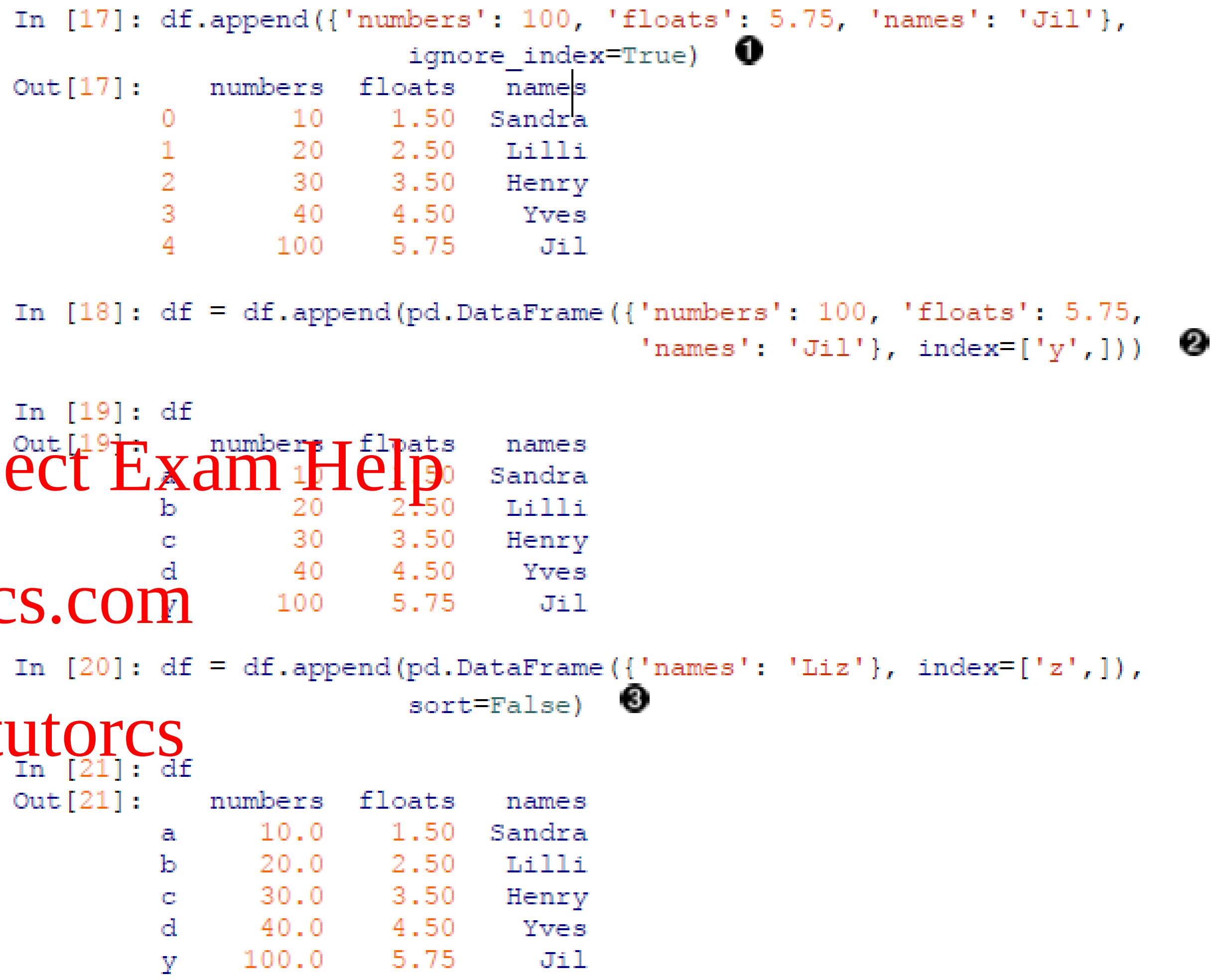
❸

Appends an incomplete data row to the `DataFrame` object, resulting in `NaN` values.

❹

Returns the different `dtypes` of the single columns; this is similar to what's possible with structured `ndarray` objects.

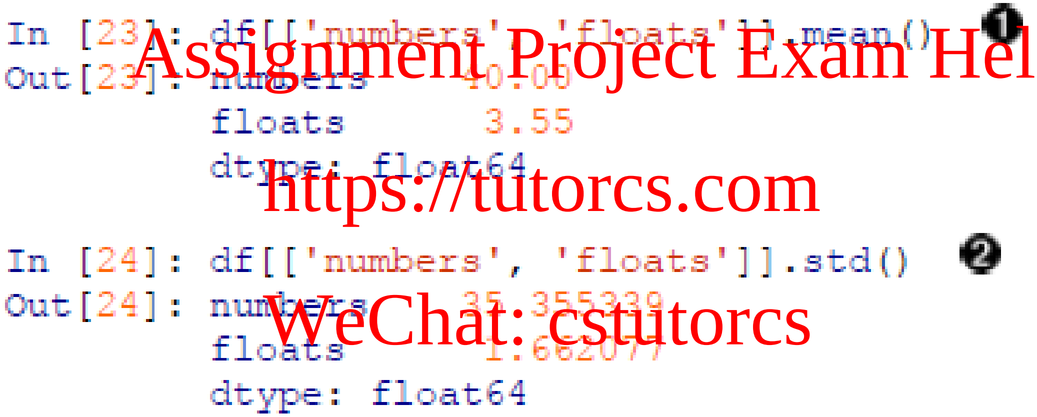
```
dtype: object
```



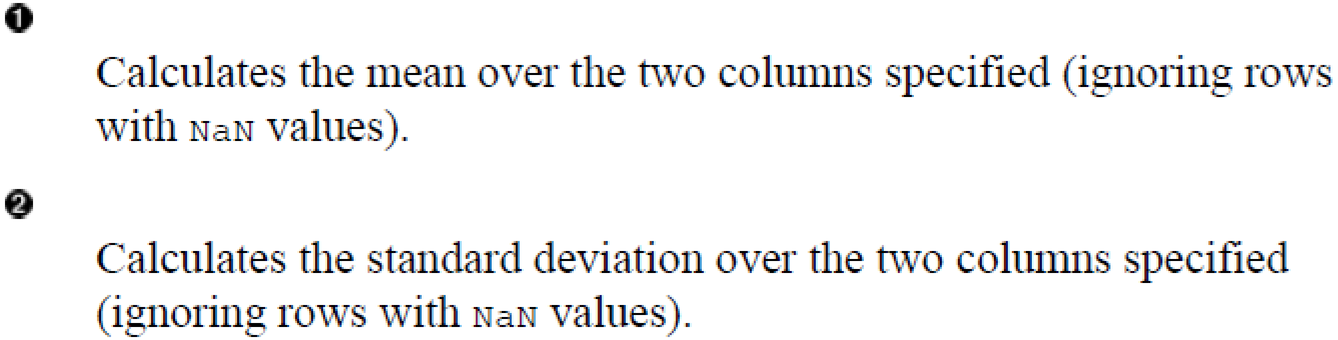
```
In [17]: df.append({'numbers': 100, 'floats': 5.75, 'names': 'Jil'},
                   ignore_index=True)  ❶
Out[17]:    numbers  floats    names
         0       10    1.50   Sandra
         1       20    2.50    Lilli
         2       30    3.50    Henry
         3       40    4.50     Yves
         4      100    5.75      Jil

In [18]: df = df.append(pd.DataFrame({'numbers': 100, 'floats': 5.75,
                                      'names': 'Jil'}, index=['y',]))  ❷

In [19]: df
Out[19]:    numbers  floats    names
         a       10    1.50   Sandra
         b       20    2.50    Lilli
         c       30    3.50    Henry
         d       40    4.50     Yves
         y      100    5.75      Jil

In [20]: df = df.append(pd.DataFrame({'names': 'Liz'}, index=['z',]),
                   sort=False)  ❸

In [21]: df
Out[21]:    numbers  floats    names
         a     10.0    1.50   Sandra
         b     20.0    2.50    Lilli
         c     30.0    3.50    Henry
         d     40.0    4.50     Yves
         y    100.0    5.75      Jil
```

First Steps with a DataFrame Class6

- Although there are now missing values, the majority of method calls will still work:



```
In [23]: df[['numbers', 'floats']].mean()  ❶
Out[23]: numbers    40.00
         floats      3.55
         dtype: float64

In [24]: df[['numbers', 'floats']].std()  ❷
Out[24]: numbers    35.355339
         floats      1.662077
         dtype: float64
```



❶

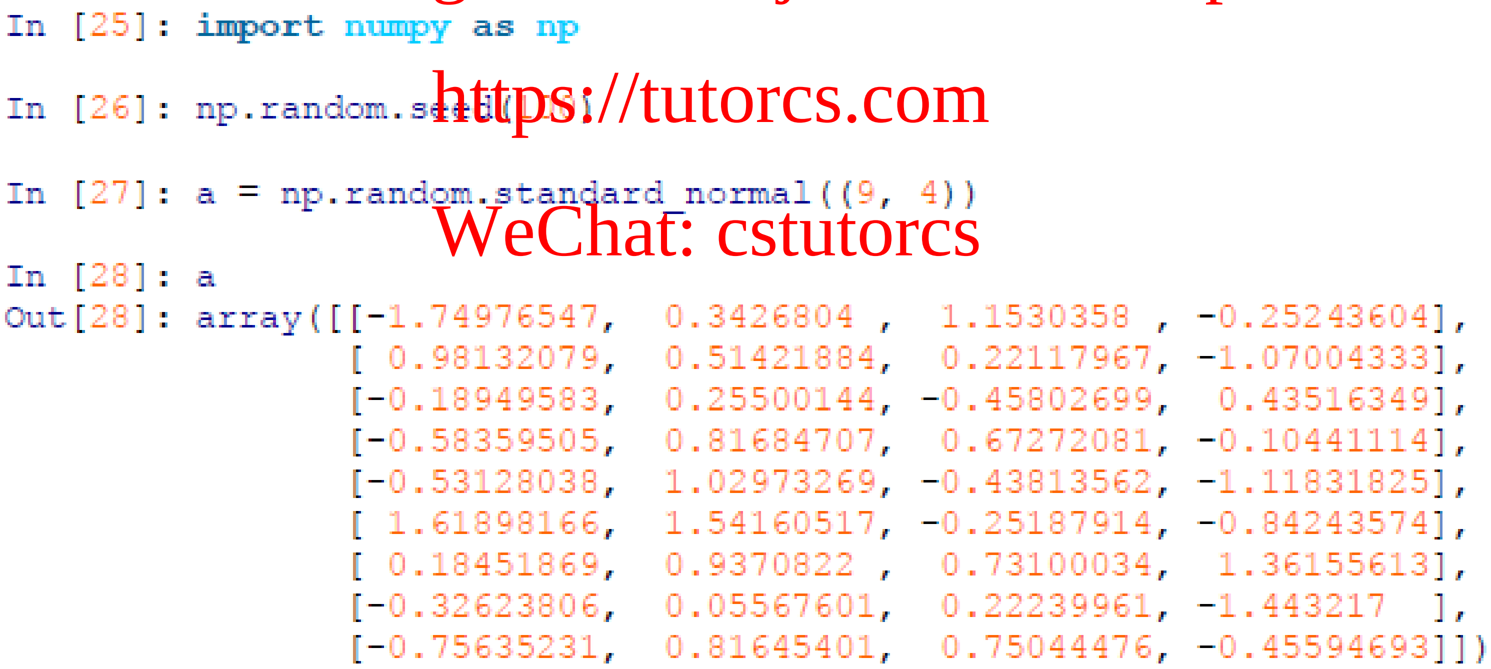
Calculates the mean over the two columns specified (ignoring rows with NaN values).

❷

Calculates the standard deviation over the two columns specified (ignoring rows with NaN values).

Second Steps with a DataFrame Class1

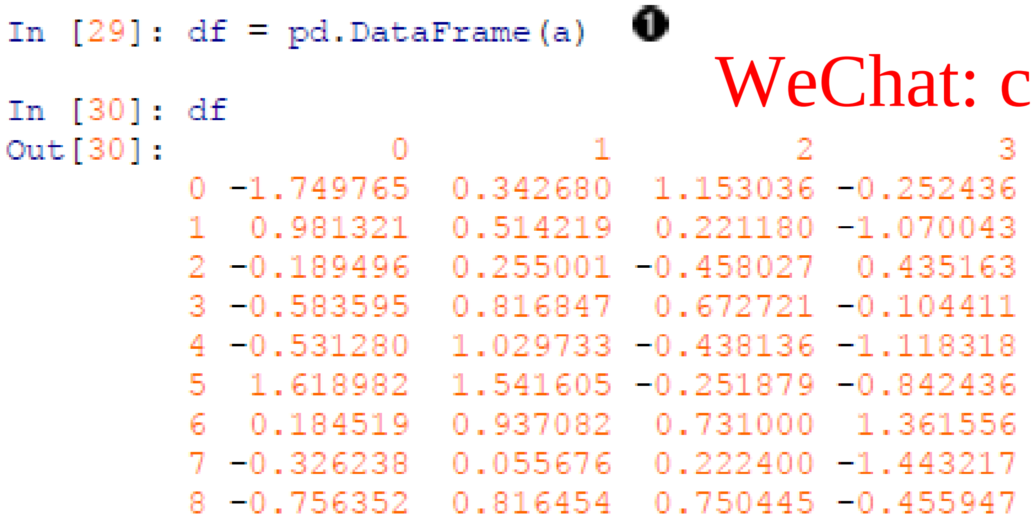
- The example in this subsection is based on an `ndarray` object with standard normally distributed random numbers.
- It explores further features such as a `DatetimeIndex` to manage time series data:



```
In [25]: import numpy as np

In [26]: np.random.seed(100)

In [27]: a = np.random.standard_normal((9, 4))

In [28]: a
Out[28]: array([[-1.74976547,  0.3426804 ,  1.1530358 , -0.25243604],
                [ 0.98132079,  0.51421884,  0.22117967, -1.07004333],
                [-0.18949583,  0.25500144, -0.45802699,  0.43516349],
                [-0.58359505,  0.81684707,  0.67272081, -0.10441114],
                [-0.53128038,  1.02973269, -0.43813562, -1.11831825],
                [ 1.61898166,  1.54160517, -0.25187914, -0.84243574],
                [ 0.18451869,  0.9370822 ,  0.73100034,  1.36155613],
                [-0.32623806,  0.05567601,  0.22239961, -1.443217  ],
                [-0.75635231,  0.81645401,  0.75044476, -0.45594693]])
```

Second Steps with a DataFrame Class2

- Although one can construct `DataFrame` objects more directly (as seen before), using an `ndarray` object is generally a good choice since `pandas` will retain the basic structure and will “only” add metainformation (e.g., index values).
- It also represents a typical use case for financial applications and scientific research in general.
- For example:



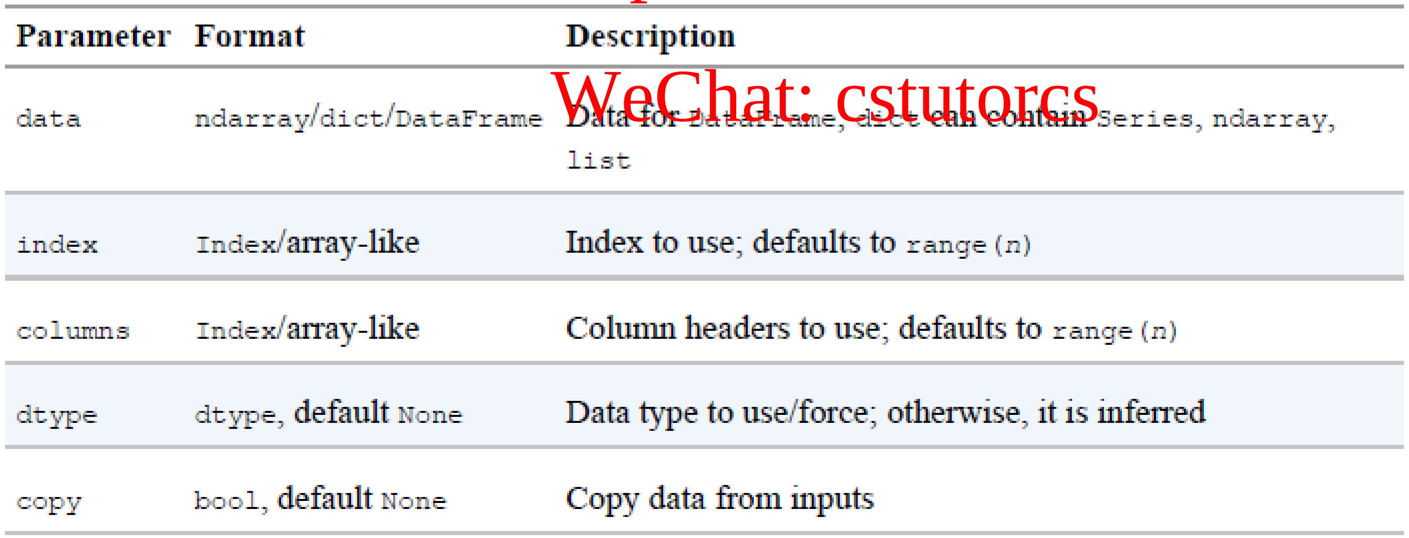
```
In [29]: df = pd.DataFrame(a)  ❶

In [30]: df
Out[30]:
          0         1         2         3
0 -1.749765  0.342680  1.153036 -0.252436
1  0.981321  0.514219  0.221180 -1.070043
2 -0.189496  0.255001 -0.458027  0.435163
3 -0.583595  0.816847  0.672721 -0.104411
4 -0.531280  1.029733 -0.438136 -1.118318
5  1.618982  1.541605 -0.251879 -0.842436
6  0.184519  0.937082  0.731000  1.361556
7 -0.326238  0.055676  0.222400 -1.443217
8 -0.756352  0.816454  0.750445 -0.455947
```

❶ Creates a `DataFrame` object from the `ndarray` object.

Second Steps with a DataFrame Class3

- Table 5-1 lists the parameters that the `DataFrame()` function takes. In the table, “array-like” means a data structure similar to an `ndarray` object — a `list`, for example. `Index` is an instance of the `pandas Index` class.

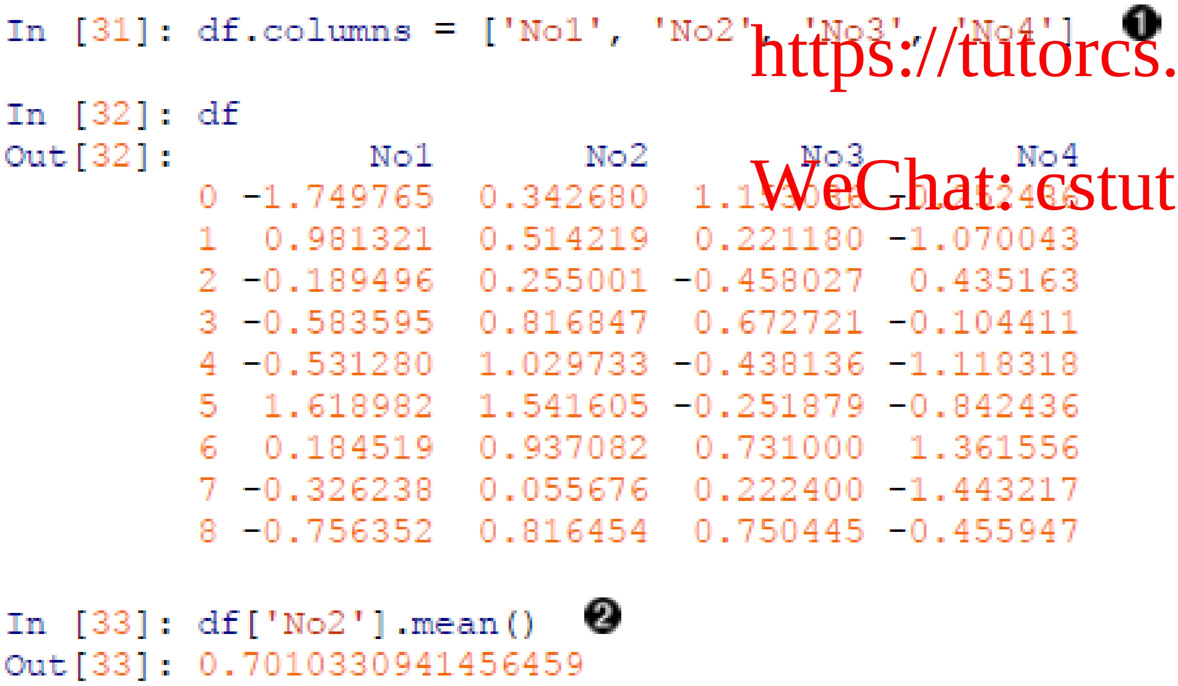
Table 5-1. Parameters of DataFrame() function



Parameter	Format	Description
`data`	`ndarray/dict/DataFrame`	Data for `DataFrame`; `dict` can contain `Series`, `ndarray`, `list`
`index`	`Index`/array-like	Index to use; defaults to `range(`*n*`)`
`columns`	`Index`/array-like	Column headers to use; defaults to `range(`*n*`)`
`dtype`	`dtype`, default `None`	Data type to use/force; otherwise, it is inferred
`copy`	`bool`, default `None`	Copy data from inputs

Second Steps with a DataFrame Class4

- As with structured arrays, and as seen before, `DataFrame` objects have column names that can be defined directly by assigning a `list` object with the right number of elements.
- This illustrates that one can define/change the attributes of the `DataFrame` object easily:



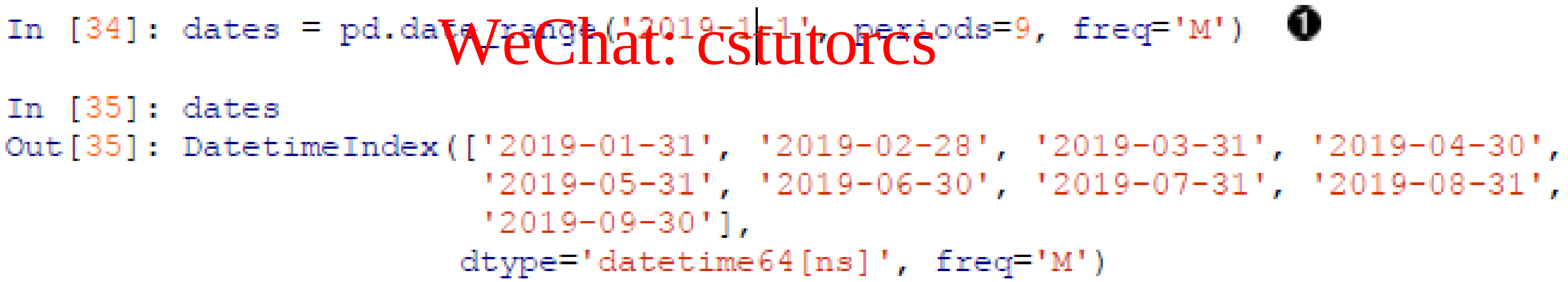
```
In [31]: df.columns = ['No1', 'No2', 'No3', 'No4']  ❶

In [32]: df
Out[32]:          No1       No2       No3       No4
         0 -1.749765  0.342680  1.153036 -0.252436
         1  0.981321  0.514219  0.221180 -1.070043
         2 -0.189496  0.255001 -0.458027  0.435163
         3 -0.583595  0.816847  0.672721 -0.104411
         4 -0.531280  1.029733 -0.438136 -1.118318
         5  1.618982  1.541605 -0.251879 -0.842436
         6  0.184519  0.937082  0.731000  1.361556
         7 -0.326238  0.055676  0.222400 -1.443217
         8 -0.756352  0.816454  0.750445 -0.455947

In [33]: df['No2'].mean()  ❷
Out[33]: 0.7010330941456459
```

❶ Specifies the column labels via a `list` object.

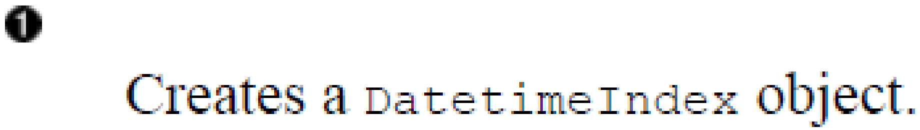
❷ Picking a column is now made easy.

Second Steps with a DataFrame Class5

- To work with financial time series data efficiently, one must be able to handle time indices well.
- This can also be considered a major strength of `pandas`.
- For example, assume that our nine data entries in the four columns correspond to month-end data, beginning in January 2019.
- A `DatetimeIndex` object is then generated with the `date_range()` function as follows:



```
In [34]: dates = pd.date_range('2019-1-1', periods=9, freq='M')  ❶

In [35]: dates
Out[35]: DatetimeIndex(['2019-01-31', '2019-02-28', '2019-03-31', '2019-04-30',
                        '2019-05-31', '2019-06-30', '2019-07-31', '2019-08-31',
                        '2019-09-30'],
                       dtype='datetime64[ns]', freq='M')
```



❶ Creates a `DatetimeIndex` object.

Second Steps with a DataFrame Class6

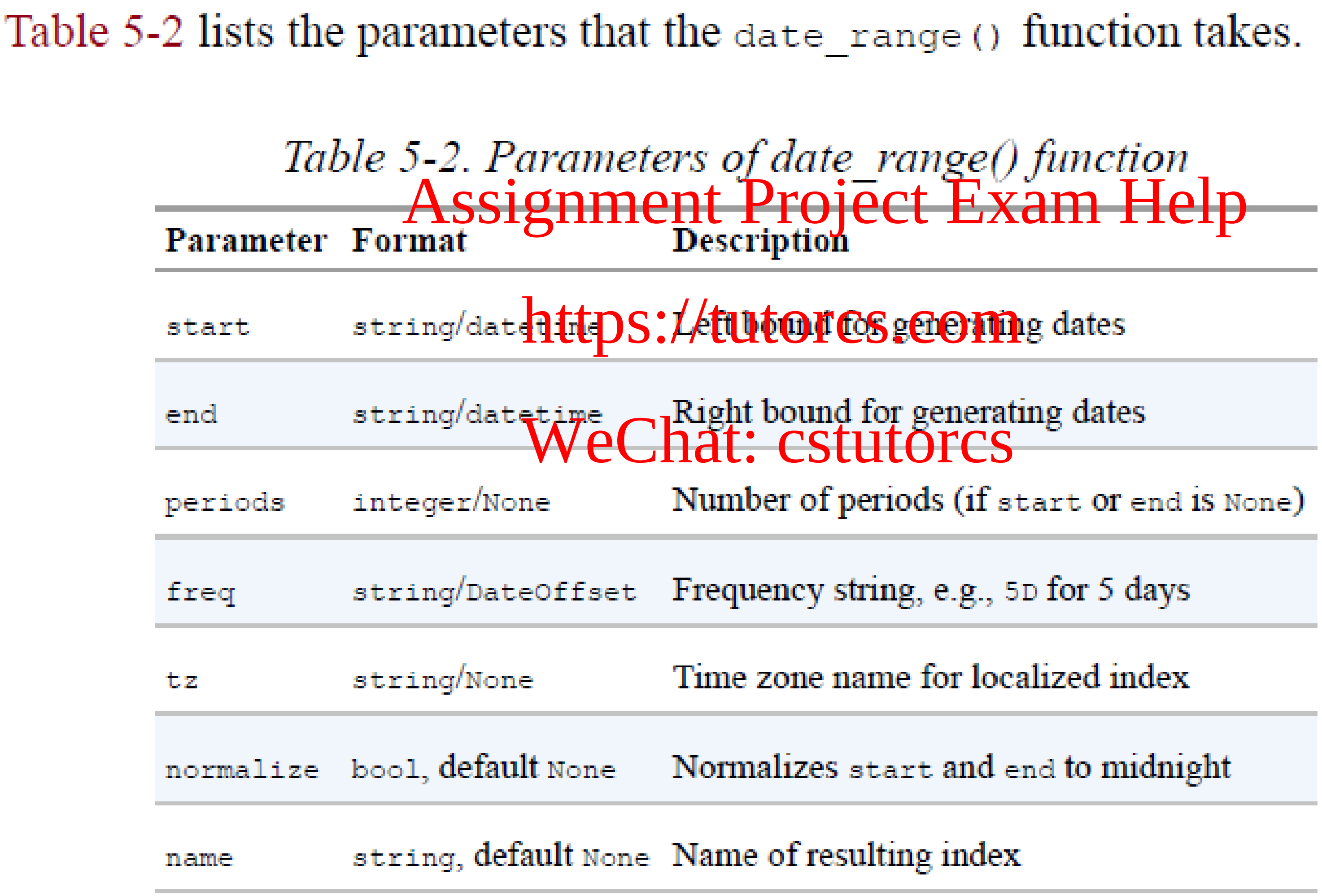


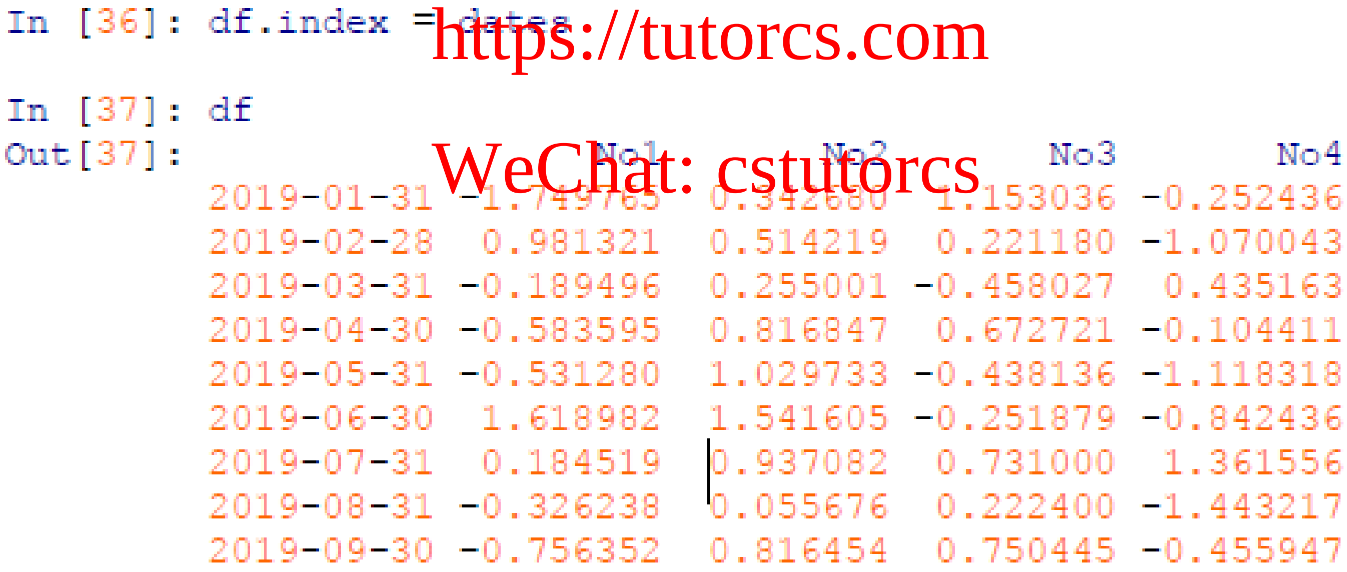
Table 5-2 lists the parameters that the `date_range()` function takes.

Table 5-2. Parameters of date_range() function

Parameter	Format	Description
`start`	`string/datetime`	Left bound for generating dates
`end`	`string/datetime`	Right bound for generating dates
`periods`	`integer/None`	Number of periods (if `start` or `end` is `None`)
`freq`	`string/DateOffset`	Frequency string, e.g., `5D` for 5 days
`tz`	`string/None`	Time zone name for localized index
`normalize`	`bool`, default `None`	Normalizes `start` and `end` to midnight
`name`	`string`, default `None`	Name of resulting index

Second Steps with a DataFrame Class7

- The following code defines the just-created `DatetimeIndex` object as the relevant index object, making a time series of the original data set:



```
In [36]: df.index = dates

In [37]: df
Out[37]:
                 No1       No2       No3       No4
2019-01-31 -1.749765  0.342680  1.153036 -0.252436
2019-02-28  0.981321  0.514219  0.221180 -1.070043
2019-03-31 -0.189496  0.255001 -0.458027  0.435163
2019-04-30 -0.583595  0.816847  0.672721 -0.104411
2019-05-31 -0.531280  1.029733 -0.438136 -1.118318
2019-06-30  1.618982  1.541605 -0.251879 -0.842436
2019-07-31  0.184519  0.937082  0.731000  1.361556
2019-08-31 -0.326238  0.055676  0.222400 -1.443217
2019-09-30 -0.756352  0.816454  0.750445 -0.455947
```

Second Steps with a DataFrame Class8

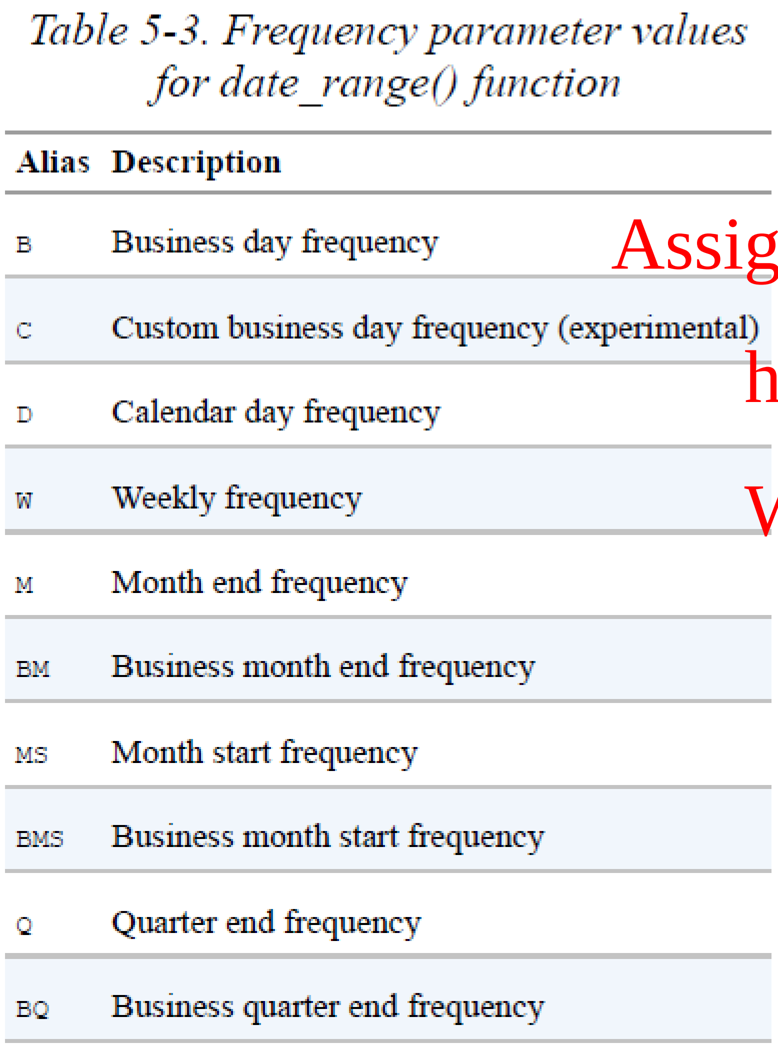
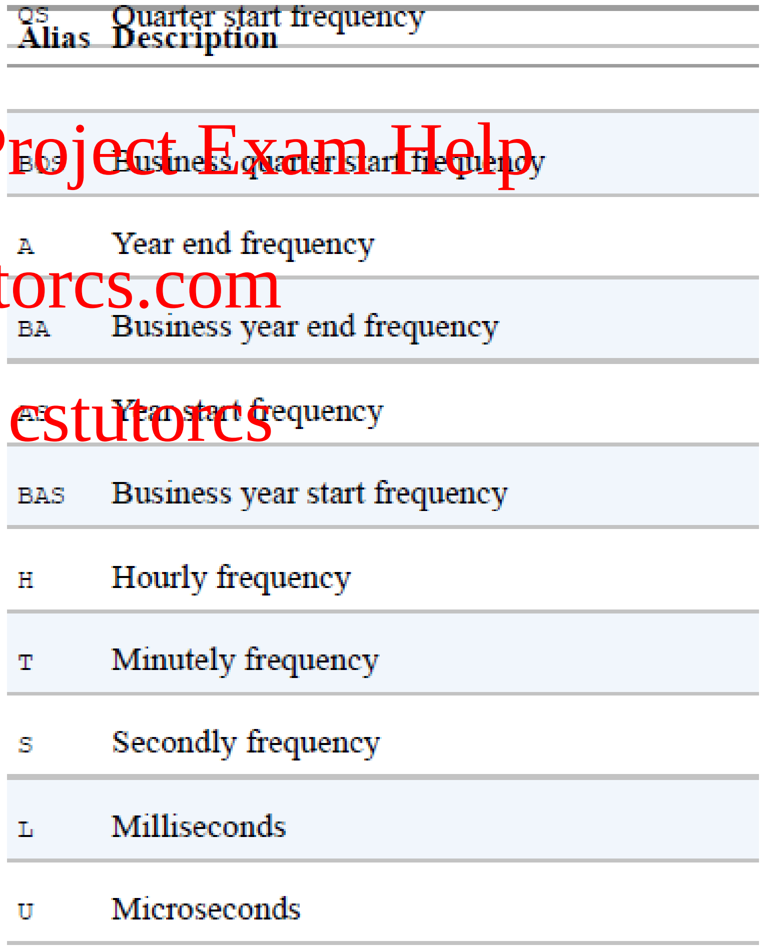


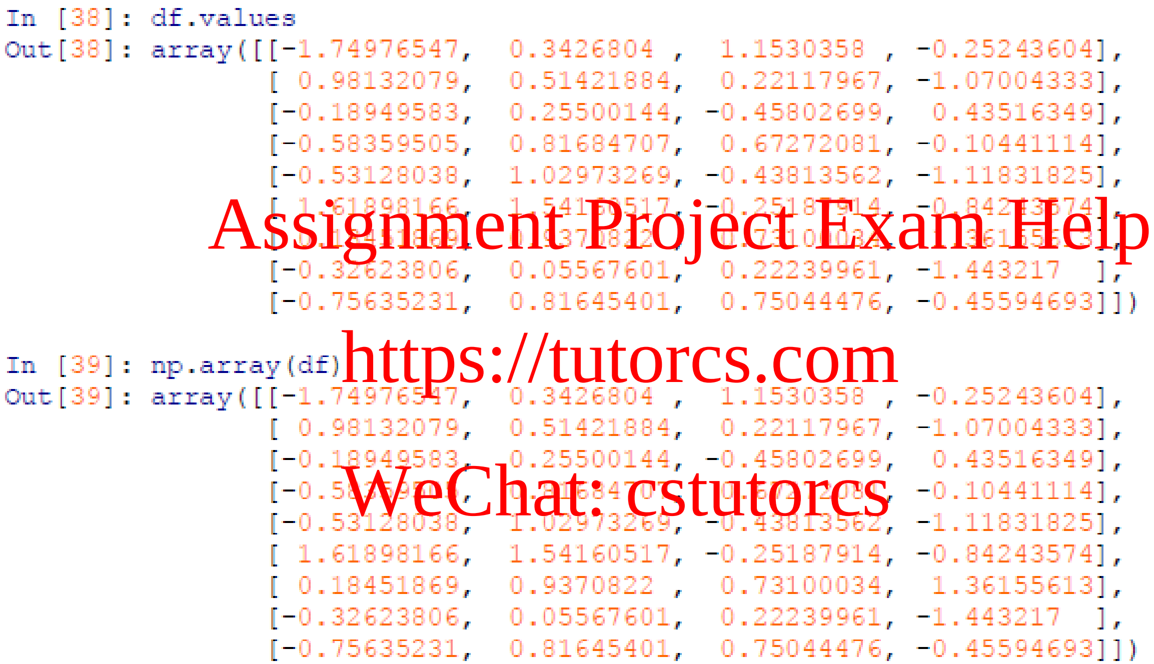
Table 5-3. Frequency parameter values for date_range() function



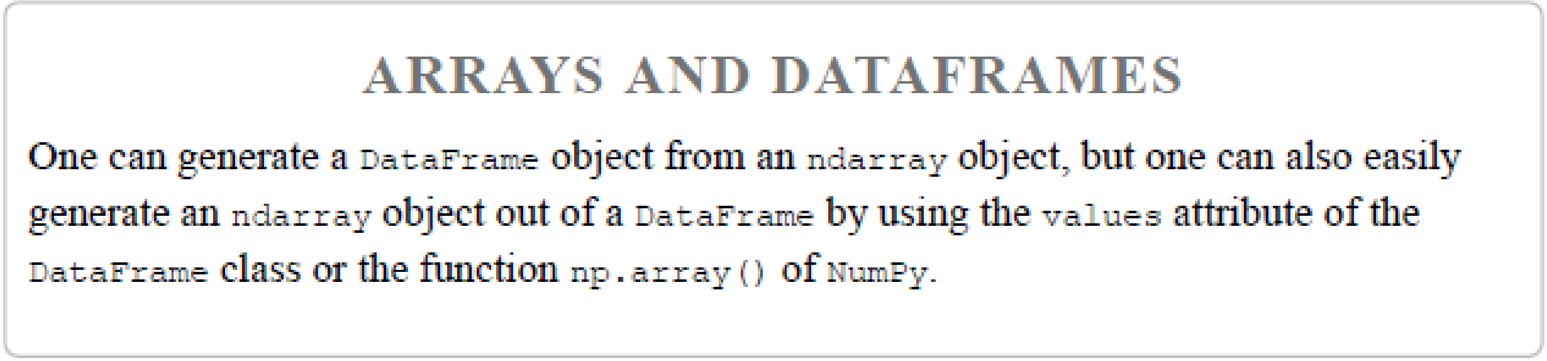
Alias	Description
B	Business day frequency
C	Custom business day frequency (experimental)
D	Calendar day frequency
W	Weekly frequency
M	Month end frequency
BM	Business month end frequency
MS	Month start frequency
BMS	Business month start frequency
Q	Quarter end frequency
BQ	Business quarter end frequency
QS	Quarter start frequency
BQS	Business quarter start frequency
A	Year end frequency
BA	Business year end frequency
AS	Year start frequency
BAS	Business year start frequency
H	Hourly frequency
T	Minutely frequency
S	Secondly frequency
L	Milliseconds
U	Microseconds

Second Steps with a DataFrame Class9

- In some circumstances, it pays off to have access to the original data set in the form of the `ndarray` object.
- The `values` attribute provides direct access to it:



```
In [38]: df.values
Out[38]: array([[-1.74976547,  0.3426804 ,  1.1530358 , -0.25243604],
                [ 0.98132079,  0.51421884,  0.22117967, -1.07004333],
                [-0.18949583,  0.25500144, -0.45802699,  0.43516349],
                [-0.58359505,  0.81684707,  0.67272081, -0.10441114],
                [-0.53128038,  1.02973269, -0.43813562, -1.11831825],
                [ 1.61898166,  1.54160517, -0.25187914, -0.84243574],
                [ 0.18451869,  0.9370822 ,  0.73100034,  1.36155613],
                [-0.32623806,  0.05567601,  0.22239961, -1.443217  ],
                [-0.75635231,  0.81645401,  0.75044476, -0.45594693]])

In [39]: np.array(df)
Out[39]: array([[-1.74976547,  0.3426804 ,  1.1530358 , -0.25243604],
                [ 0.98132079,  0.51421884,  0.22117967, -1.07004333],
                [-0.18949583,  0.25500144, -0.45802699,  0.43516349],
                [-0.58359505,  0.81684707,  0.67272081, -0.10441114],
                [-0.53128038,  1.02973269, -0.43813562, -1.11831825],
                [ 1.61898166,  1.54160517, -0.25187914, -0.84243574],
                [ 0.18451869,  0.9370822 ,  0.73100034,  1.36155613],
                [-0.32623806,  0.05567601,  0.22239961, -1.443217  ],
                [-0.75635231,  0.81645401,  0.75044476, -0.45594693]])
```

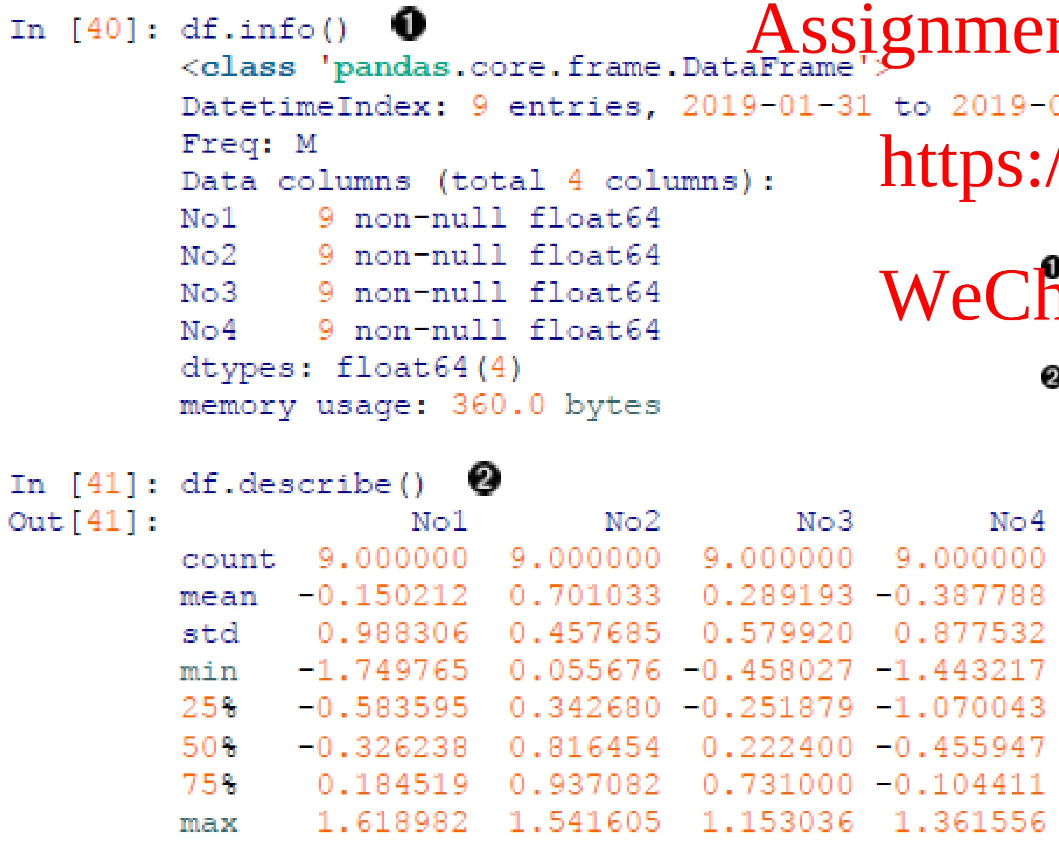


ARRAYS AND DATAFRAMES

One can generate a `DataFrame` object from an `ndarray` object, but one can also easily generate an `ndarray` object out of a `DataFrame` by using the `values` attribute of the `DataFrame` class or the function `np.array()` of `NumPy`.

Basic Analytics1

- Like the `NumPy ndarray` class, the `pandas DataFrame` class has a multitude of convenience methods built in.
- As a starter, consider the methods `info()` and `describe()`:



```
In [40]: df.info()  ❶
         <class 'pandas.core.frame.DataFrame'>
         DatetimeIndex: 9 entries, 2019-01-31 to 2019-09-30
         Freq: M
         Data columns (total 4 columns):
         No1     9 non-null float64
         No2     9 non-null float64
         No3     9 non-null float64
         No4     9 non-null float64
         dtypes: float64(4)
         memory usage: 360.0 bytes

In [41]: df.describe()  ❷
Out[41]:             No1        No2        No3        No4
         count  9.000000   9.000000   9.000000   9.000000
         mean  -0.150212   0.701033   0.289193  -0.387788
         std    0.988306   0.457685   0.579920   0.877532
         min   -1.749765   0.055676  -0.458027  -1.443217
         25%   -0.583595   0.342680  -0.251879  -1.070043
         50%   -0.326238   0.816454   0.222400  -0.455947
         75%    0.184519   0.937082   0.731000  -0.104411
         max    1.618982   1.541605   1.153036   1.361556
```

This method prints information about a DataFrame including the index dtype and columns, non-null values and memory usage. Note the number of cells in each column (non-null values).

❶ Provides metainformation regarding the data, columns, and index.

❷ Provides helpful summary statistics per column (for numerical data).

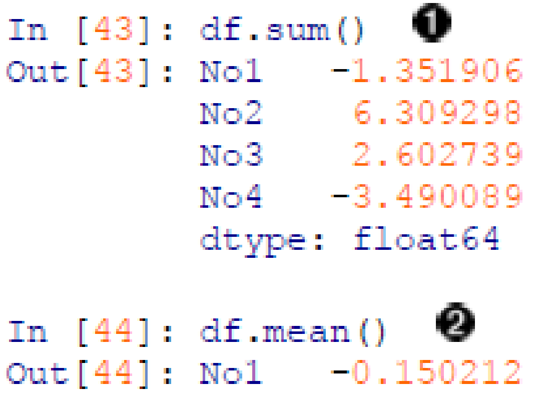
25th Percentile - Also known as the first, or lower, quartile. The 25th percentile is the value at which 25% of the answers lie below that value, and 75% of the answers lie above that value.

50th Percentile - Also known as the Median. The median cuts the data set in half. Half of the answers lie below the median and half lie above the median.

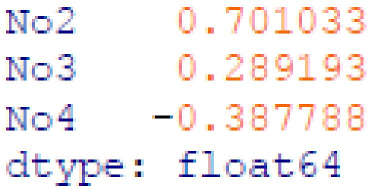
75th Percentile - Also known as the third, or upper, quartile. The 75th percentile is the value at which 25% of the answers lie above that value and 75% of the answers lie below that value.

Basic Analytics2

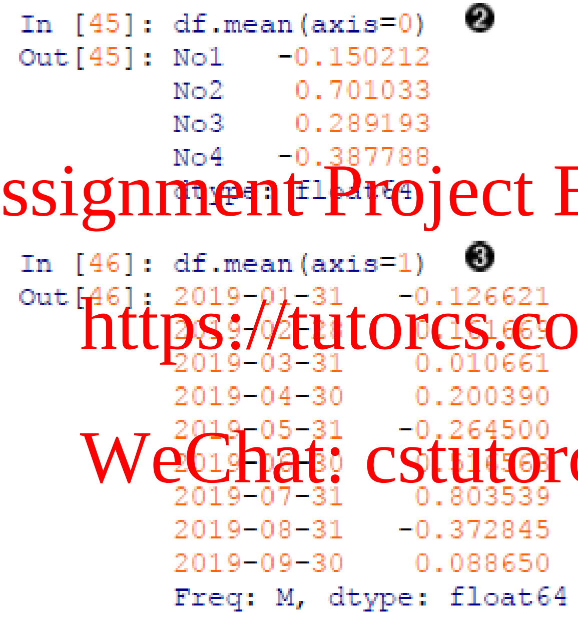
- In addition, one can easily get the column-wise or row-wise sums, means, and cumulative sums:



```
In [43]: df.sum() ❶
Out[43]: No1   -1.351906
         No2    6.309298
         No3    2.602739
         No4   -3.490089
         dtype: float64

In [44]: df.mean() ❷
Out[44]: No1   -0.150212
```



```
No2    0.701033
No3    0.289193
No4   -0.387788
dtype: float64
```



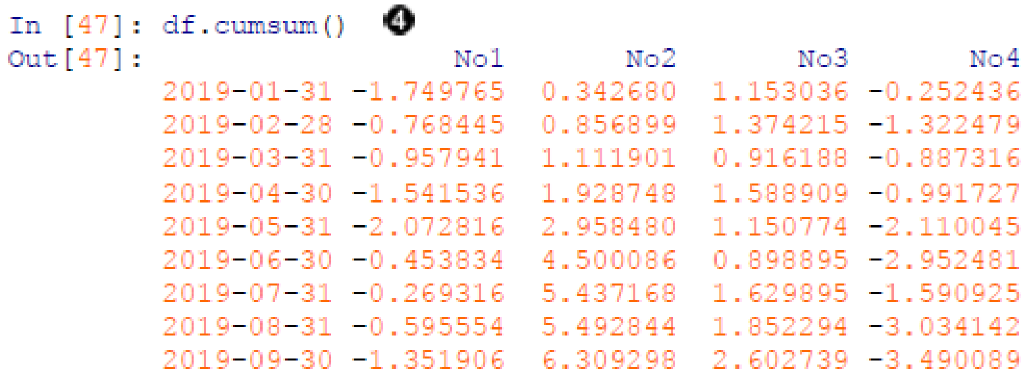
```
In [45]: df.mean(axis=0) ❷
Out[45]: No1   -0.150212
         No2    0.701033
         No3    0.289193
         No4   -0.387788
         dtype: float64

In [46]: df.mean(axis=1) ❸
Out[46]: 2019-01-31   -0.126621
         2019-02-28    0.161669
         2019-03-31    0.010661
         2019-04-30    0.200390
         2019-05-31   -0.264500
         2019-06-30    0.516568
         2019-07-31    0.803539
         2019-08-31   -0.372845
         2019-09-30    0.088650
         Freq: M, dtype: float64
```

❶ Column-wise sum.

❷ Column-wise mean.

❸ Row-wise mean.

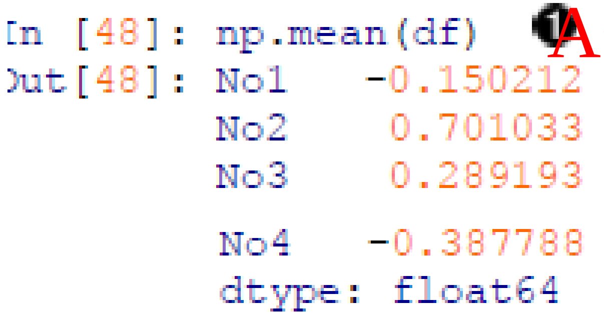
❹ Column-wise cumulative sum (starting at first index position).



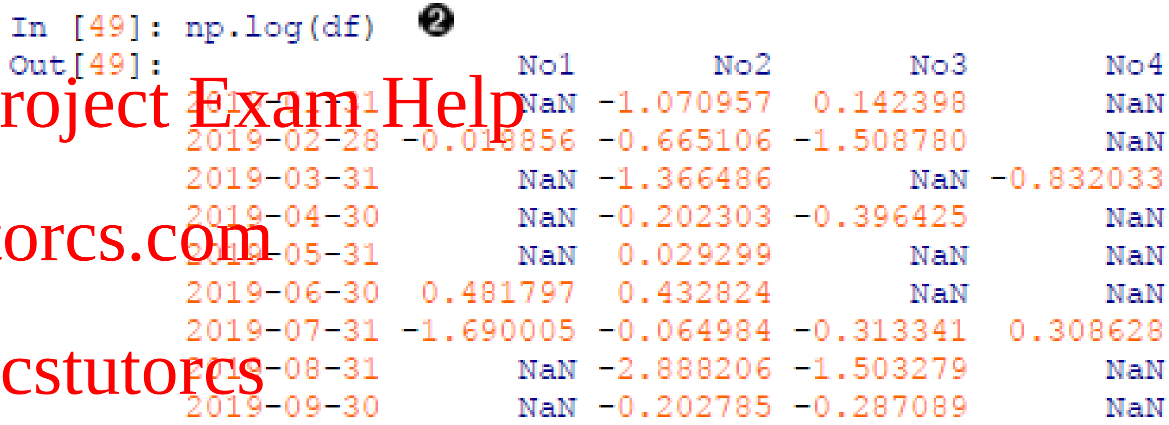
```
In [47]: df.cumsum() ❹
Out[47]:                  No1       No2       No3       No4
         2019-01-31 -1.749765  0.342680  1.153036 -0.252436
         2019-02-28 -0.768445  0.856899  1.374215 -1.322479
         2019-03-31 -0.957941  1.111901  0.916188 -0.887316
         2019-04-30 -1.541536  1.928748  1.588909 -0.991727
         2019-05-31 -2.072816  2.958480  1.150774 -2.110045
         2019-06-30 -0.453834  4.500086  0.898895 -2.952481
         2019-07-31 -0.269316  5.437168  1.629895 -1.590925
         2019-08-31 -0.595554  5.492844  1.852294 -3.034142
         2019-09-30 -1.351906  6.309298  2.602739 -3.490089
```

Basic Analytics3

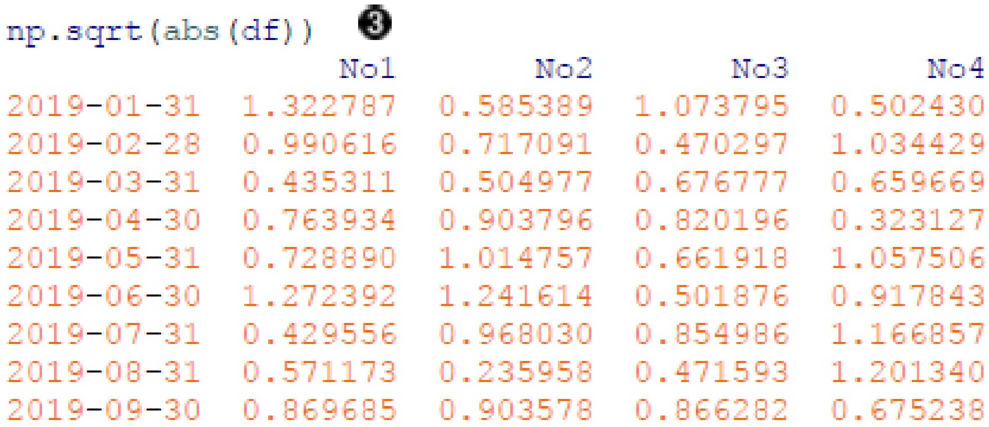
- `DataFrame` objects also understand `NumPy` universal functions, as expected:



```
In [48]: np.mean(df) ❶
Out[48]: No1   -0.150212
         No2    0.701033
         No3    0.289193
         No4   -0.387788
         dtype: float64
```



```
In [49]: np.log(df) ❷
Out[49]:
                 No1       No2       No3       No4
2019-01-31       NaN -1.070957  0.142398       NaN
2019-02-28 -0.018856 -0.665106 -1.508780       NaN
2019-03-31       NaN -1.366486       NaN -0.832033
2019-04-30       NaN -0.202303 -0.396425       NaN
2019-05-31       NaN  0.029299       NaN       NaN
2019-06-30  0.481797  0.432824       NaN       NaN
2019-07-31 -1.690005 -0.064984 -0.313341  0.308628
2019-08-31       NaN -2.888206 -1.503279       NaN
2019-09-30       NaN -0.202785 -0.287089       NaN
```



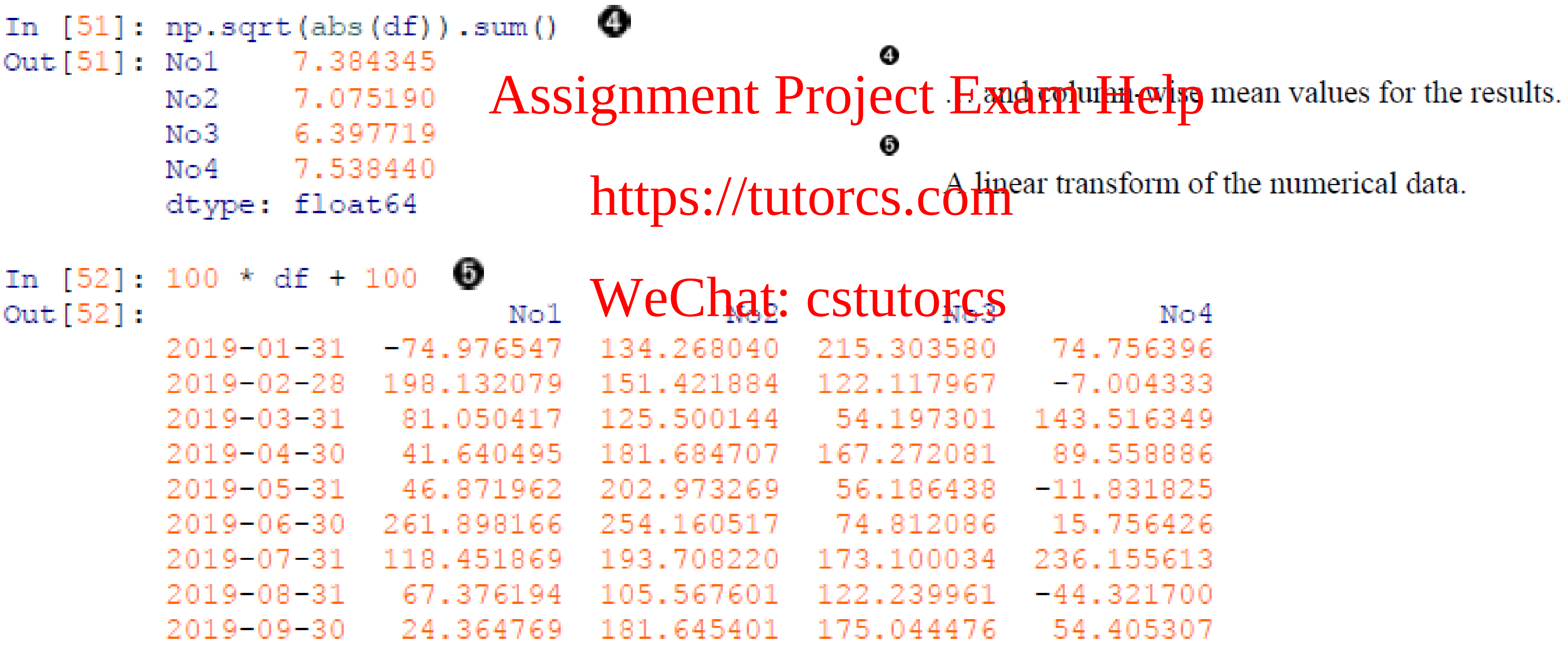
```
np.sqrt(abs(df)) ❸
                 No1       No2       No3       No4
2019-01-31  1.322787  0.585389  1.073795  0.502430
2019-02-28  0.990616  0.717091  0.470297  1.034429
2019-03-31  0.435311  0.504977  0.676777  0.659669
2019-04-30  0.763934  0.903796  0.820196  0.323127
2019-05-31  0.728890  1.014757  0.661918  1.057506
2019-06-30  1.272392  1.241614  0.501876  0.917843
2019-07-31  0.429556  0.968030  0.854986  1.166857
2019-08-31  0.571173  0.235958  0.471593  1.201340
2019-09-30  0.869685  0.903578  0.866282  0.675238
```

❶ Column-wise mean.

❷ Element-wise natural logarithm; a warning is raised but the calculation runs through, resulting in multiple `NaN` values.

❸ Element-wise square root for the absolute values …

Basic Analytics4



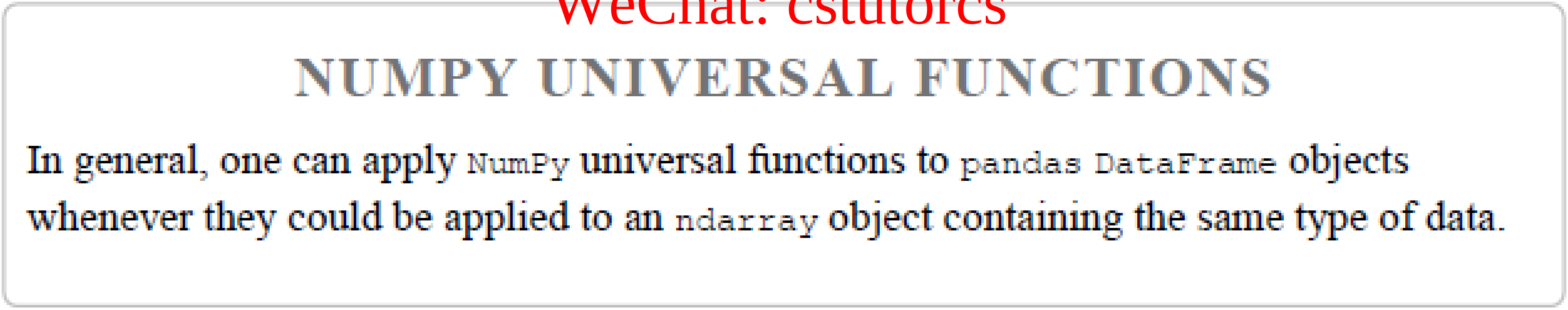
```
In [51]: np.sqrt(abs(df)).sum() ❹
Out[51]: No1    7.384345
         No2    7.075190
         No3    6.397719
         No4    7.538440
         dtype: float64

In [52]: 100 * df + 100 ❺
Out[52]:
                   No1         No2         No3         No4
         2019-01-31  -74.976547  134.268040  215.303580   74.756396
         2019-02-28  198.132079  151.421884  122.117967   -7.004333
         2019-03-31   81.050417  125.500144   54.197301  143.516349
         2019-04-30   41.640495  181.684707  167.272081   89.558886
         2019-05-31   46.871962  202.973269   56.186438  -11.831825
         2019-06-30  261.898166  254.160517   74.812086   15.756426
         2019-07-31  118.451869  193.708220  173.100034  236.155613
         2019-08-31   67.376194  105.567601  122.239961  -44.321700
         2019-09-30   24.364769  181.645401  175.044476   54.405307
```

❹ … and column-wise mean values for the results.

❺ A linear transform of the numerical data.

Basic Analytics5

- `pandas` is quite error tolerant, in the sense that it captures errors and just puts a `NaN` value where the respective mathematical operation fails.
- Not only this, but as briefly shown before, one can also work with such incomplete data sets as if they were complete in a number of cases.
- This comes in handy, since reality is characterized by incomplete data sets more often than one might wish.



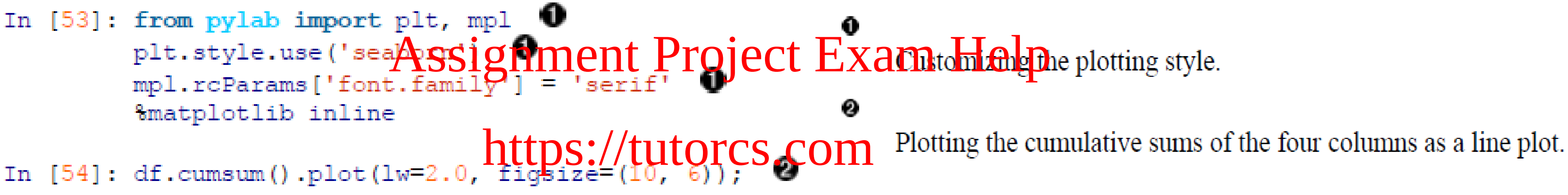
NUMPY UNIVERSAL FUNCTIONS

In general, one can apply `NumPy` universal functions to `pandas` `DataFrame` objects whenever they could be applied to an `ndarray` object containing the same type of data.

- A numpy.array is a function that returns a numpy.ndarray.
- Anumpy.ndarray() is a class, while numpy.array() is a method / function to create ndarray

Basic Visualisation1: Introduction

- Plotting of data is only one line of code away in general, once the data is stored in a `DataFrame` object (see Figure 5-1):



```
In [53]: from pylab import plt, mpl ❶
         plt.style.use('seaborn') ❶
         mpl.rcParams['font.family'] = 'serif' ❶
         %matplotlib inline

In [54]: df.cumsum().plot(lw=2.0, figsize=(10, 6)); ❷
```

❶ Customizing the plotting style.

❷ Plotting the cumulative sums of the four columns as a line plot.

- Basically, `pandas` provides a wrapper around `matplotlib` (see Chapter 7), specifically designed for `DataFrame` objects.
- Table 5-4 lists the parameters that the `plot()` method takes.

Basic Visualisation2: Line plot example

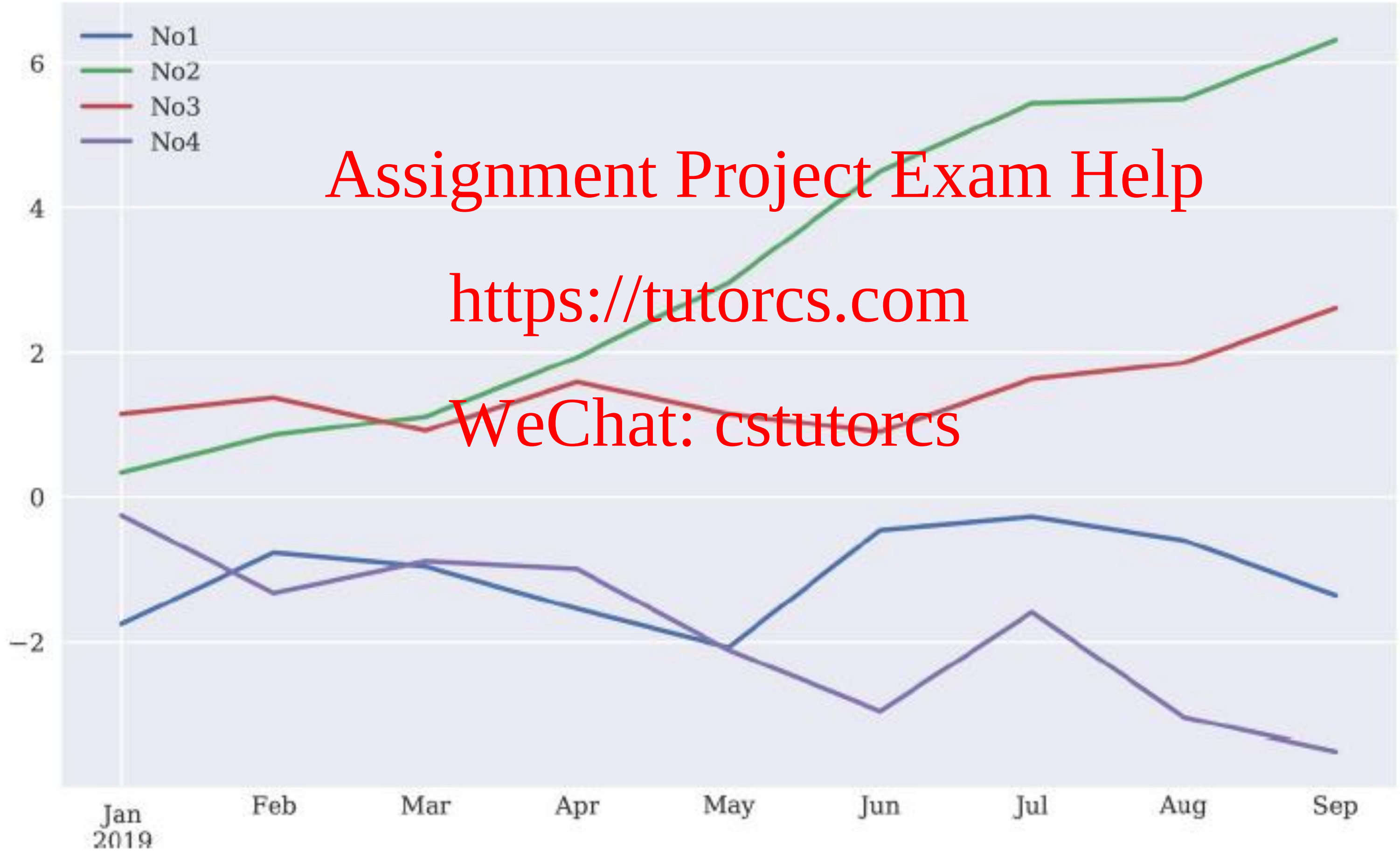


Figure 5-1. Line plot of a DataFrame object

Basic Visualisation3: Plot methods

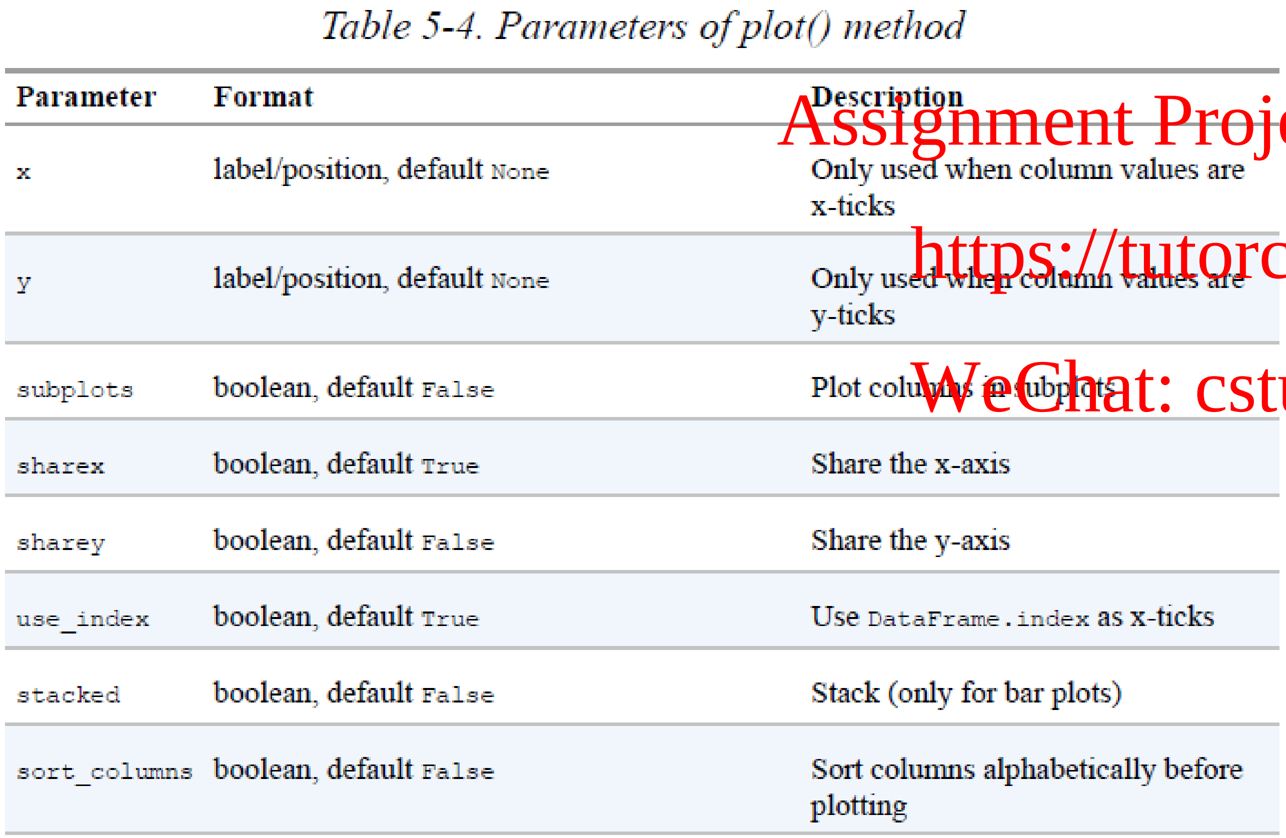
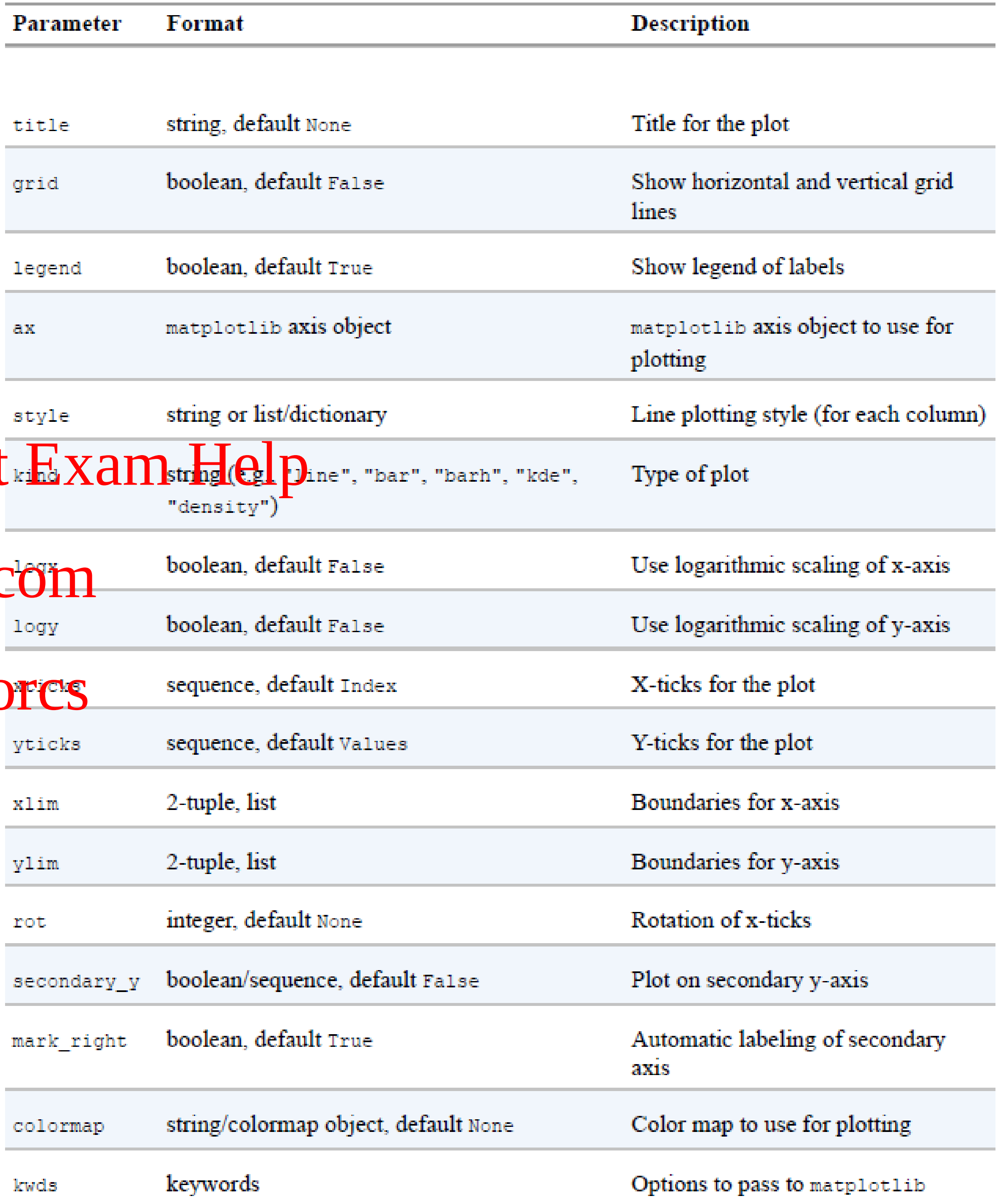


Table 5-4. Parameters of plot() method

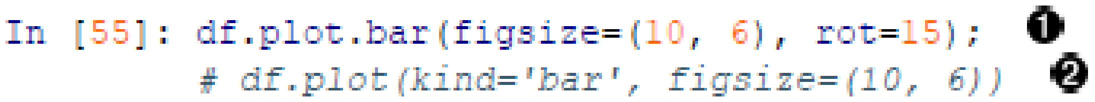
Parameter	Format	Description
`x`	label/position, default `None`	Only used when column values are x-ticks
`y`	label/position, default `None`	Only used when column values are y-ticks
`subplots`	boolean, default `False`	Plot columns in subplots
`sharex`	boolean, default `True`	Share the x-axis
`sharey`	boolean, default `False`	Share the y-axis
`use_index`	boolean, default `True`	Use `DataFrame.index` as x-ticks
`stacked`	boolean, default `False`	Stack (only for bar plots)
`sort_columns`	boolean, default `False`	Sort columns alphabetically before plotting



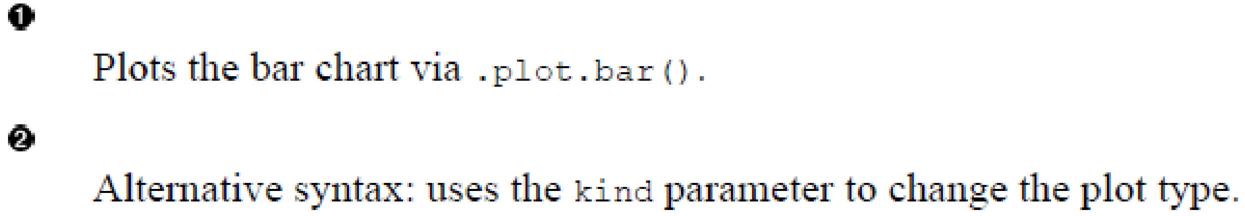
Parameter	Format	Description
`title`	string, default `None`	Title for the plot
`grid`	boolean, default `False`	Show horizontal and vertical grid lines
`legend`	boolean, default `True`	Show legend of labels
`ax`	`matplotlib` axis object	`matplotlib` axis object to use for plotting
`style`	string or list/dictionary	Line plotting style (for each column)
`kind`	string (e.g., `"line"`, `"bar"`, `"barh"`, `"kde"`, `"density"`)	Type of plot
`logx`	boolean, default `False`	Use logarithmic scaling of x-axis
`logy`	boolean, default `False`	Use logarithmic scaling of y-axis
`xticks`	sequence, default `Index`	X-ticks for the plot
`yticks`	sequence, default `Values`	Y-ticks for the plot
`xlim`	2-tuple, list	Boundaries for x-axis
`ylim`	2-tuple, list	Boundaries for y-axis
`rot`	integer, default `None`	Rotation of x-ticks
`secondary_y`	boolean/sequence, default `False`	Plot on secondary y-axis
`mark_right`	boolean, default `True`	Automatic labeling of secondary axis
`colormap`	string/colormap object, default `None`	Color map to use for plotting
`kwds`	keywords	Options to pass to `matplotlib`

Basic Visualisation4: Bar plot example

- As another example, consider a bar plot of the same data (see Figure 5-2):



```
In [55]: df.plot.bar(figsize=(10, 6), rot=15);  ❶
         # df.plot(kind='bar', figsize=(10, 6))  ❷
```



❶ Plots the bar chart via `.plot.bar()`.

❷ Alternative syntax: uses the `kind` parameter to change the plot type.

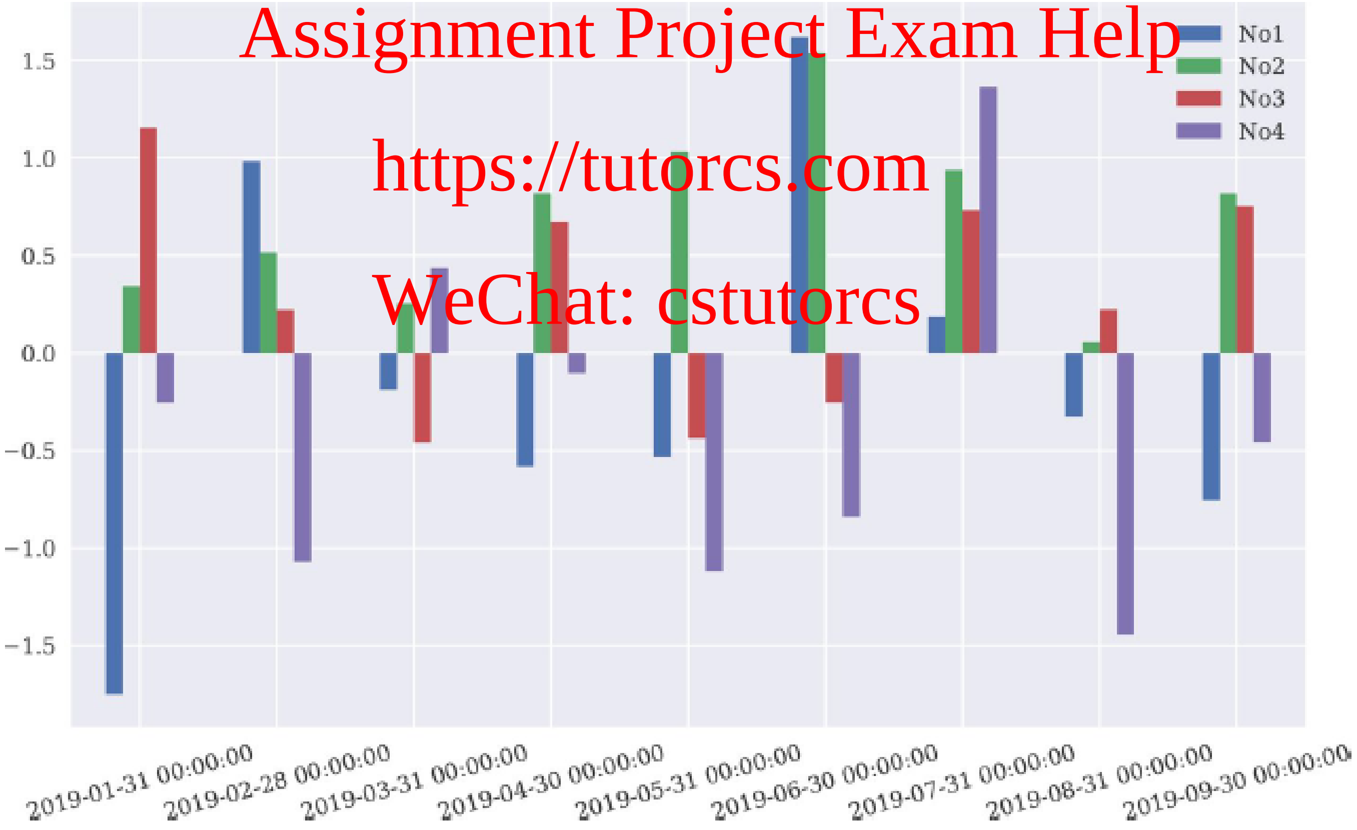
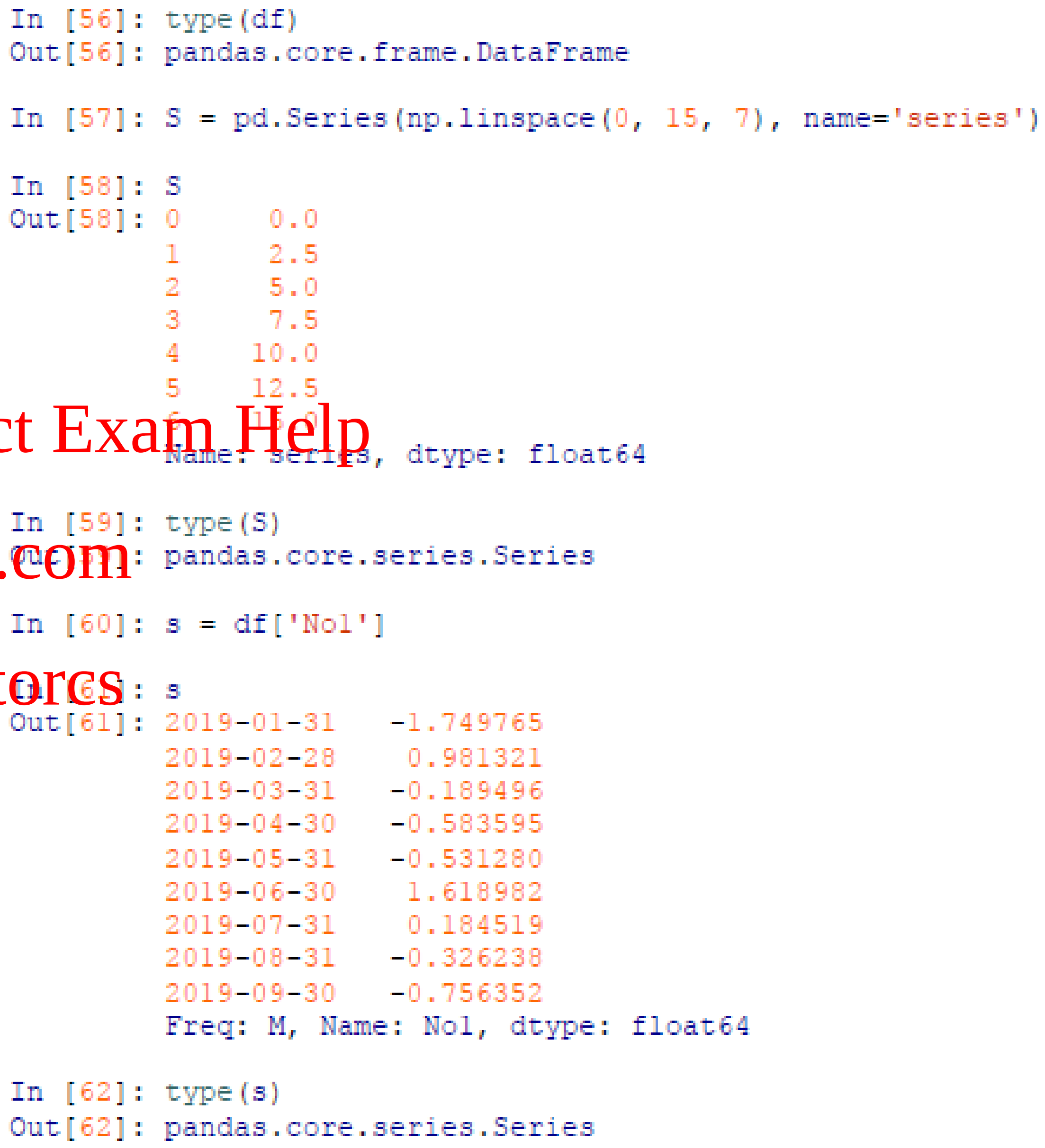


Figure 5-2. Bar plot of a DataFrame object

The Series Class1

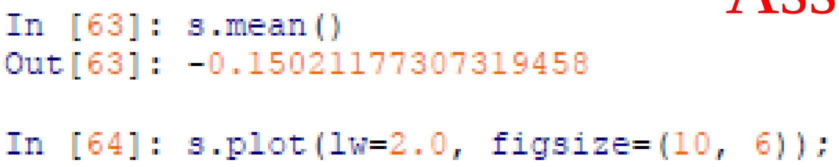
- So far this chapter has worked mainly with the `pandas DataFrame` class.
- `Series` is another important class that comes with `pandas`. It is characterized by the fact that it has only a single column of data.
- In that sense, it is a specialization of the `DataFrame` class that shares many but not all of its characteristics and capabilities.
- A `Series` object is obtained when a single column is selected from a multicolumn `DataFrame` object:



```
In [56]: type(df)
Out[56]: pandas.core.frame.DataFrame

In [57]: S = pd.Series(np.linspace(0, 15, 7), name='series')

In [58]: S
Out[58]: 0     0.0
         1     2.5
         2     5.0
         3     7.5
         4    10.0
         5    12.5
         6    15.0
         Name: series, dtype: float64

In [59]: type(S)
Out[59]: pandas.core.series.Series

In [60]: s = df['No1']

In [61]: s
Out[61]: 2019-01-31   -1.749765
         2019-02-28    0.981321
         2019-03-31   -0.189496
         2019-04-30   -0.583595
         2019-05-31   -0.531280
         2019-06-30    1.618982
         2019-07-31    0.184519
         2019-08-31   -0.326238
         2019-09-30   -0.756352
         Freq: M, Name: No1, dtype: float64

In [62]: type(s)
Out[62]: pandas.core.series.Series
```

The Series Class2

- The main `DataFrame` methods are available for `Series` objects as well. For illustration, consider the `mean()` and `plot()` methods (see Figure 5-3):



```
In [63]: s.mean()
Out[63]: -0.15021177307319458

In [64]: s.plot(lw=2.0, figsize=(10, 6));
```

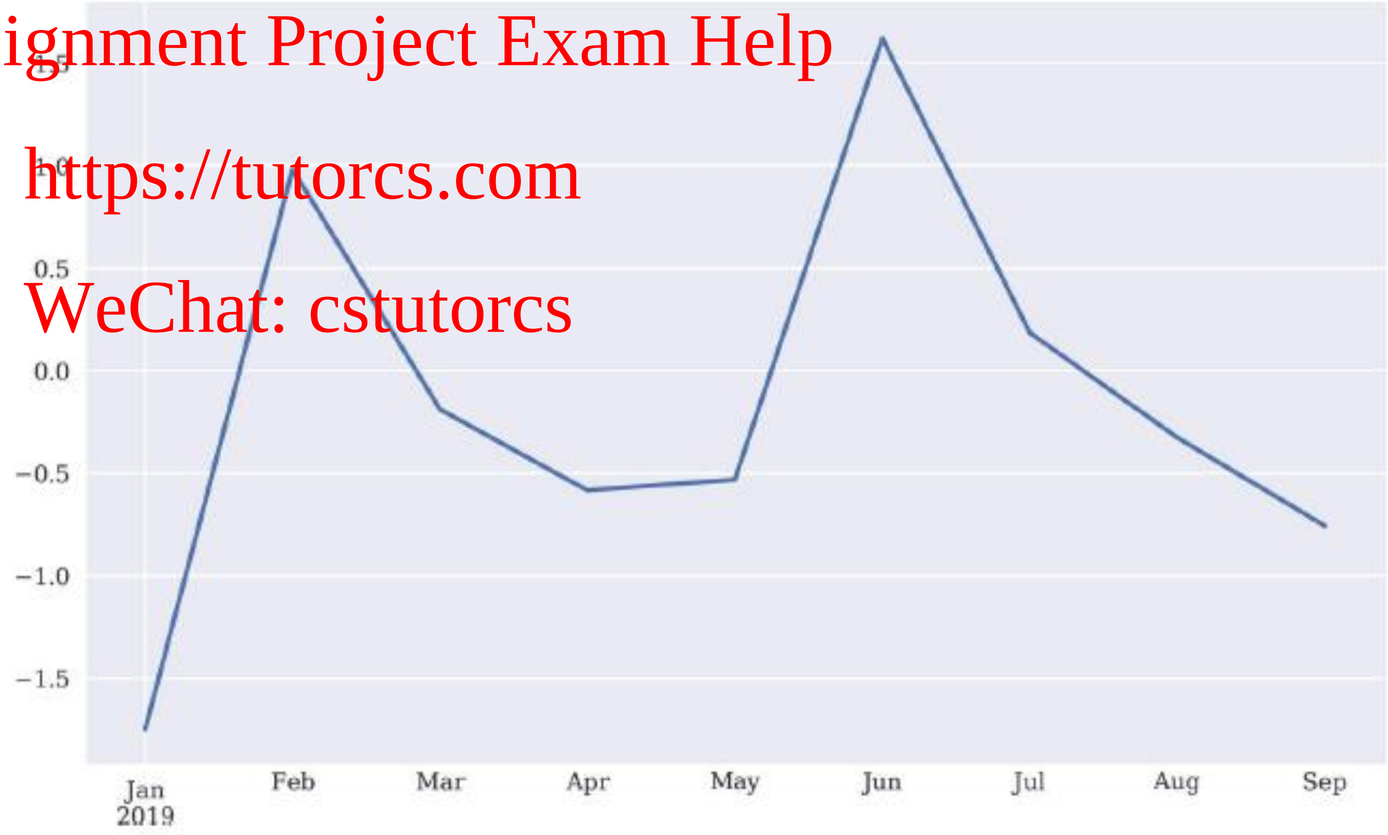
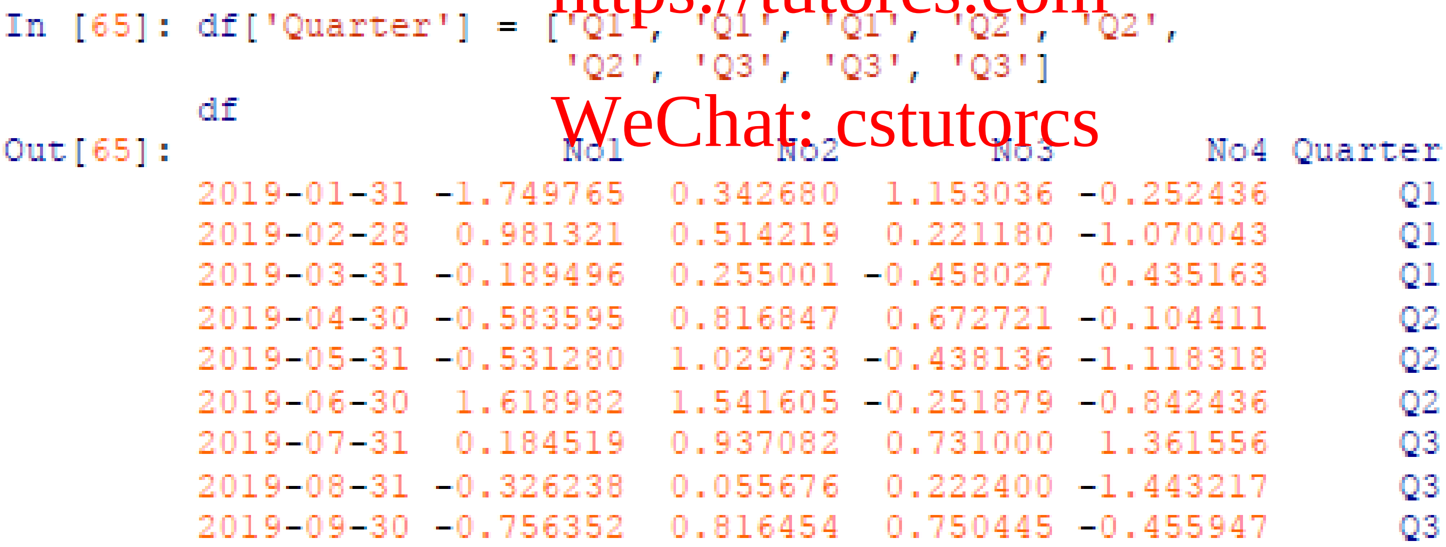


Figure 5-3. Line plot of a Series object

GroupBy Operations1

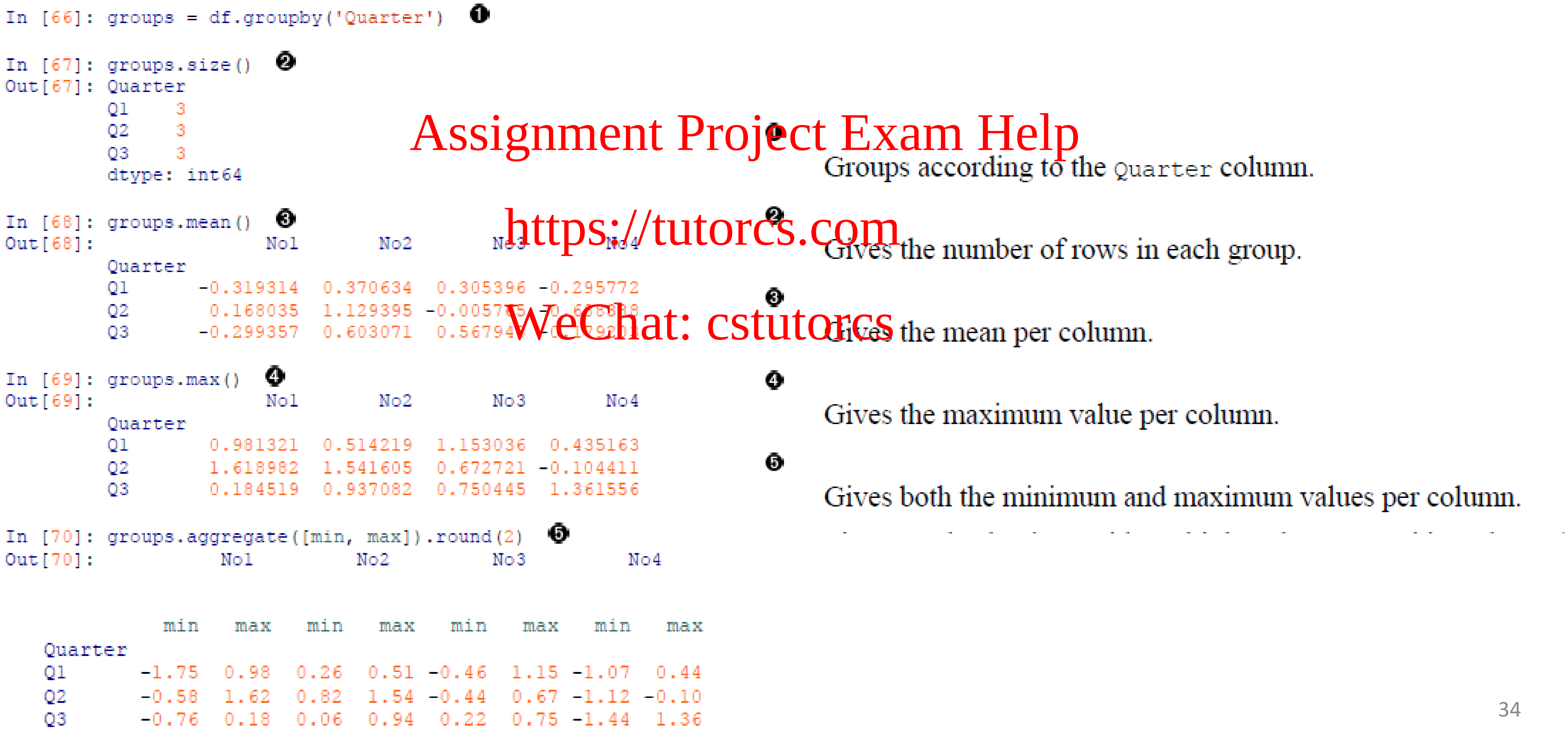
- `pandas` has powerful and flexible grouping capabilities.
- They work similarly to grouping in SQL as well as pivot tables in Microsoft Excel.
- To have something to group by one can add, for instance, a column indicating the quarter the respective data of the index belongs to:



```
In [65]: df['Quarter'] = ['Q1', 'Q1', 'Q1', 'Q2', 'Q2',
                          'Q2', 'Q3', 'Q3', 'Q3']

         df
Out[65]:                  No1       No2       No3       No4 Quarter
         2019-01-31 -1.749765  0.342680  1.153036 -0.252436      Q1
         2019-02-28  0.981321  0.514219  0.221180 -1.070043      Q1
         2019-03-31 -0.189496  0.255001 -0.458027  0.435163      Q1
         2019-04-30 -0.583595  0.816847  0.672721 -0.104411      Q2
         2019-05-31 -0.531280  1.029733 -0.438136 -1.118318      Q2
         2019-06-30  1.618982  1.541605 -0.251879 -0.842436      Q2
         2019-07-31  0.184519  0.937082  0.731000  1.361556      Q3
         2019-08-31 -0.326238  0.055676  0.222400 -1.443217      Q3
         2019-09-30 -0.756352  0.816454  0.750445 -0.455947      Q3
```

GroupBy Operations2

- The following code groups by the `Quarter` column and outputs statistics for the single groups:



```
In [66]: groups = df.groupby('Quarter')  ❶

In [67]: groups.size()  ❷
Out[67]: Quarter
         Q1    3
         Q2    3
         Q3    3
         dtype: int64

In [68]: groups.mean()  ❸
Out[68]:               No1       No2       No3       No4
         Quarter
         Q1      -0.319314  0.370634  0.305396 -0.295772
         Q2       0.168035  1.129395 -0.005765 -0.668288
         Q3      -0.299357  0.603071  0.567945 -0.179208

In [69]: groups.max()  ❹
Out[69]:               No1       No2       No3       No4
         Quarter
         Q1       0.981321  0.514219  1.153036  0.435163
         Q2       1.618982  1.541605  0.672721 -0.104411
         Q3       0.184519  0.937082  0.750445  1.361556

In [70]: groups.aggregate([min, max]).round(2)  ❺
Out[70]:          No1          No2          No3          No4


              min   max   min   max   min   max   min   max
   Quarter
   Q1       -1.75  0.98  0.26  0.51 -0.46  1.15 -1.07  0.44
   Q2       -0.58  1.62  0.82  1.54 -0.44  0.67 -1.12 -0.10
   Q3       -0.76  0.18  0.06  0.94  0.22  0.75 -1.44  1.36
```

❶ Groups according to the `Quarter` column.

❷ Gives the number of rows in each group.

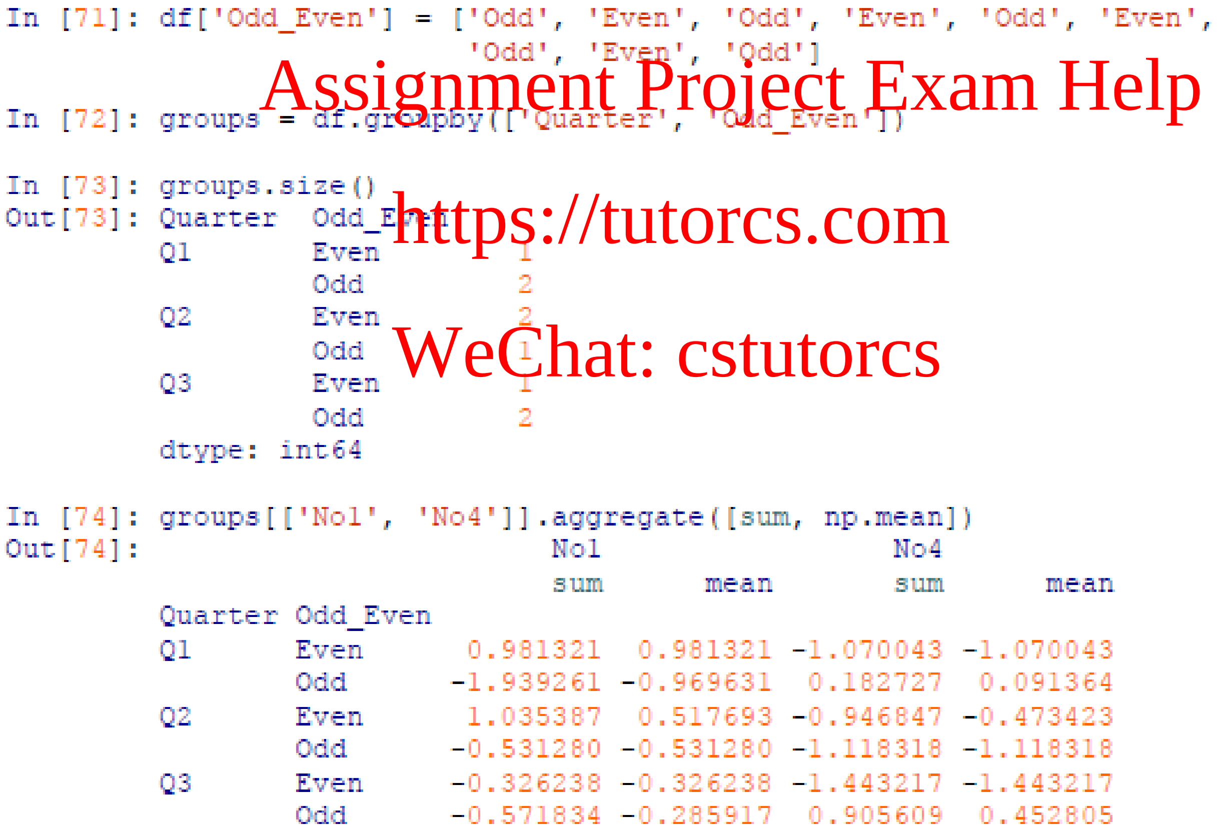
❸ Gives the mean per column.

❹ Gives the maximum value per column.

❺ Gives both the minimum and maximum values per column.

GroupBy Operations3

- Grouping can also be done with multiple columns.
- To this end, another column, indicating whether the month of the index date is odd or even, is introduced:



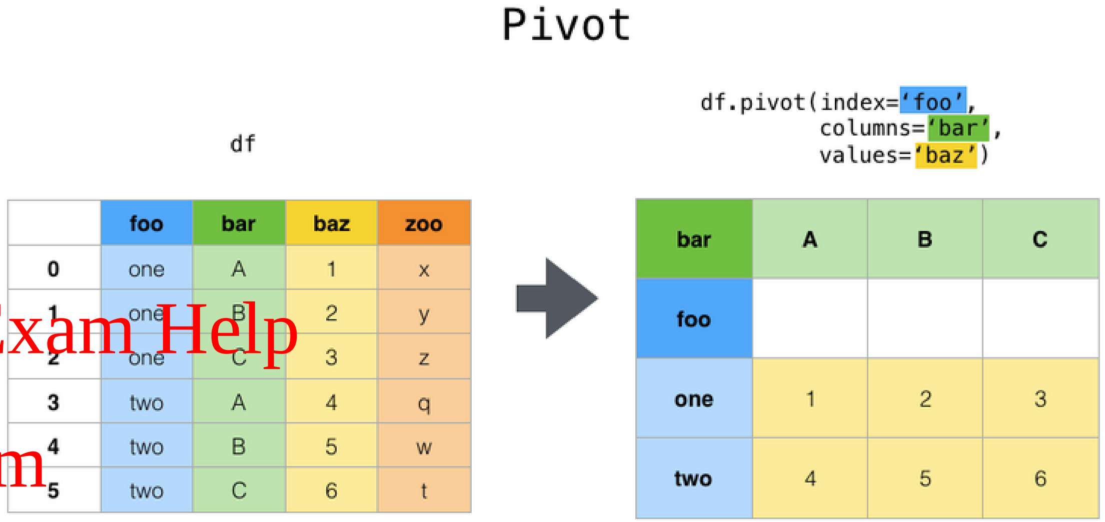
```
In [71]: df['Odd_Even'] = ['Odd', 'Even', 'Odd', 'Even', 'Odd', 'Even',
                           'Odd', 'Even', 'Odd']

In [72]: groups = df.groupby(['Quarter', 'Odd_Even'])

In [73]: groups.size()
Out[73]: Quarter  Odd_Even
         Q1       Even        1
                  Odd         2
         Q2       Even        2
                  Odd         1
         Q3       Even        1
                  Odd         2
         dtype: int64

In [74]: groups[['No1', 'No4']].aggregate([sum, np.mean])
Out[74]:                       No1                 No4
                               sum      mean       sum      mean
         Quarter Odd_Even
         Q1      Even     0.981321  0.981321 -1.070043 -1.070043
                 Odd     -1.939261 -0.969631  0.182727  0.091364
         Q2      Even     1.035387  0.517693 -0.946847 -0.473423
                 Odd     -0.531280 -0.531280 -1.118318 -1.118318
         Q3      Even    -0.326238 -0.326238 -1.443217 -1.443217
                 Odd     -0.571834 -0.285917  0.905609  0.452805
```

- This concludes the introduction to `pandas` and the use of `DataFrame` objects.

*Panel Data Pivot Table1

Create a spreadsheet-style pivot table as a DataFrame.

The levels in the pivot table will be stored in MultiIndex objects (hierarchical indexes) on the index and columns of the result DataFrame.



Pivot

```
df.pivot(index='foo',
         columns='bar',
         values='baz')
```

df

	foo	bar	baz	zoo
0	one	A	1	x
1	one	B	2	y
2	one	C	3	z
3	two	A	4	q
4	two	B	5	w
5	two	C	6	t

bar	A	B	C
foo			
one	1	2	3
two	4	5	6

Function pivot_table()

- pandas.pivot_table(data, values=None, index=None, columns=None, aggfunc='mean', fill_value=None, margins=False, dropna=True, margins_name='All', observed=False)
- Create a spreadsheet-style pivot table as a DataFrame. The levels in the pivot table will be stored in MultiIndex objects (hierarchical indexes) on the index and columns of the result DataFrame

Function pivot()

- DataFrame.pivot(self, index=None, columns=None, values=None)
- Reshape data (produce a "pivot" table) based on column values. Uses unique values from specified *index* / *columns* to form axes of the resulting DataFrame. This function does not support data aggregation, multiple values will result in a MultiIndex in the columns.

Function transpose()

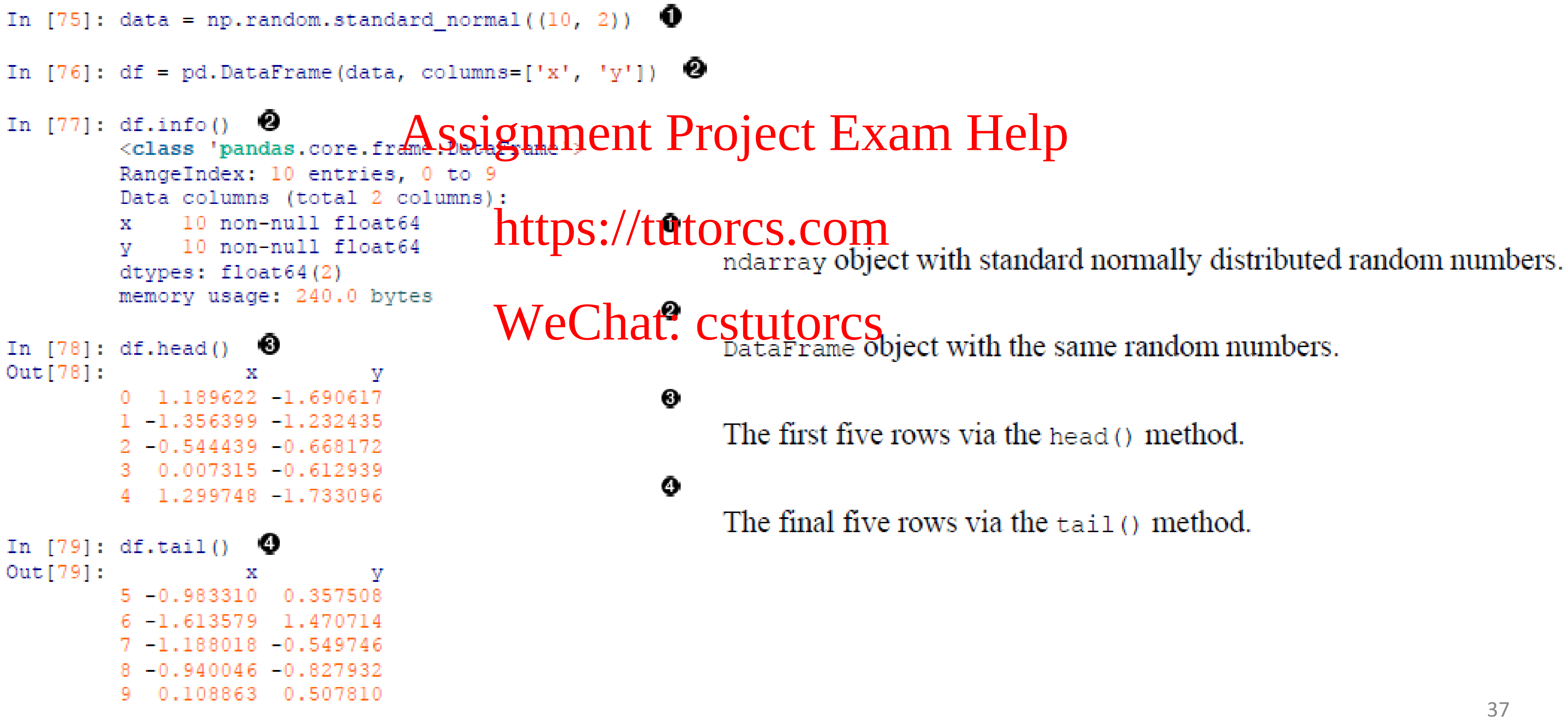
- DataFrame.transpose(self, *args, **kwargs)
- Reflect the DataFrame over its main diagonal by writing rows as columns and vice-versa.

Resources:

- https://pandas.pydata.org/pandas-docs/stable/reference/api/pandas.pivot_table.html
- https://jakevdp.github.io/PythonDataScienceHandbook/03.09-pivot-tables.html
- https://www.geeksforgeeks.org/python-pandas-pivot_table/
- https://www.geeksforgeeks.org/python-pandas-dataframe-transpose/

Complex Selection1

- Often, data selection is accomplished by formulation of conditions on column values, and potentially combining multiple such conditions logically.
- Consider the following data set:



```
In [75]: data = np.random.standard_normal((10, 2))  ❶

In [76]: df = pd.DataFrame(data, columns=['x', 'y'])  ❷

In [77]: df.info()  ❷
         <class 'pandas.core.frame.DataFrame'>
         RangeIndex: 10 entries, 0 to 9
         Data columns (total 2 columns):
         x    10 non-null float64
         y    10 non-null float64
         dtypes: float64(2)
         memory usage: 240.0 bytes

In [78]: df.head()  ❸
Out[78]:
                 x         y
         0  1.189622 -1.690617
         1 -1.356399 -1.232435
         2 -0.544439 -0.668172
         3  0.007315 -0.612939
         4  1.299748 -1.733096

In [79]: df.tail()  ❹
Out[79]:
                 x         y
         5 -0.983310  0.357508
         6 -1.613579  1.470714
         7 -1.188018 -0.549746
         8 -0.940046 -0.827932
         9  0.108863  0.507810
```

❶ ndarray object with standard normally distributed random numbers.

❷ DataFrame object with the same random numbers.

❸ The first five rows via the head() method.

❹ The final five rows via the tail() method.

Complex Selection2

- The following code illustrates the application of Python’s comparison operators and logical operators on values in the two columns:

❶

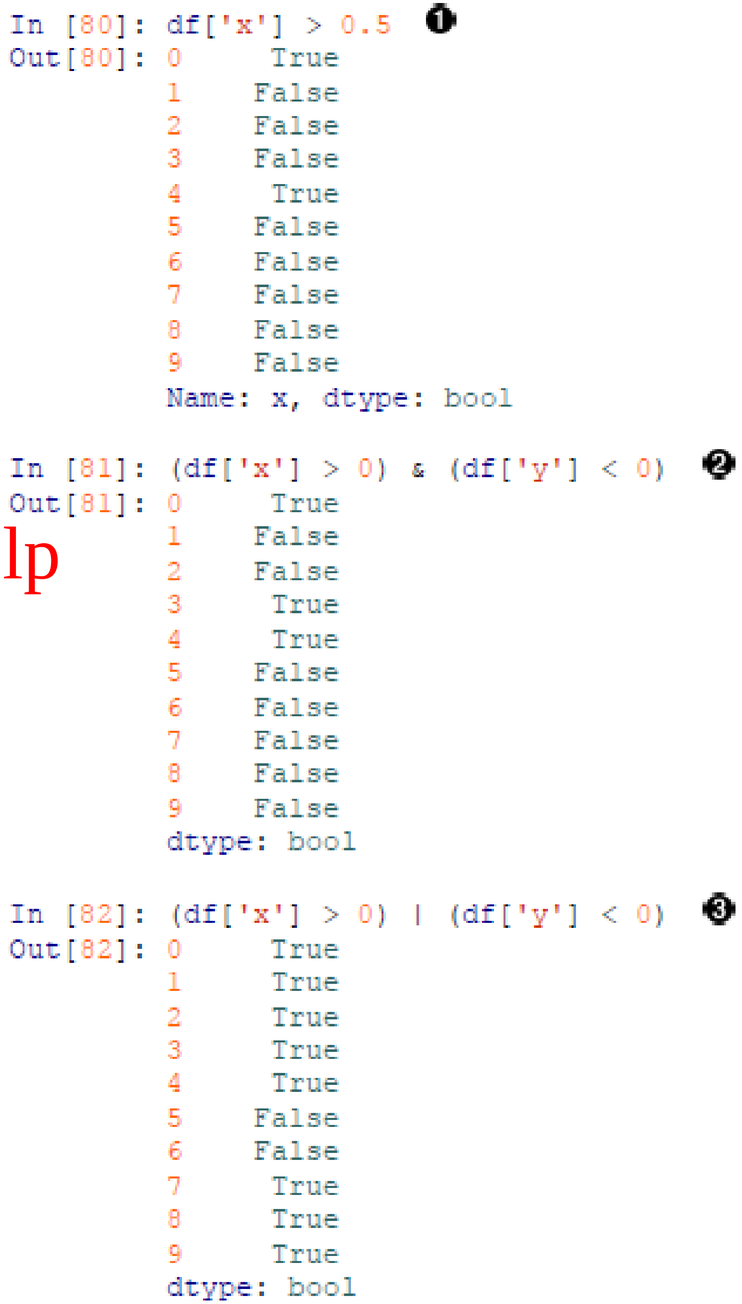
Check whether value in column x is greater than 0.5.

❷

Check whether value in column x is positive *and* value in column y is negative.

❸

Check whether value in column x is positive *or* value in column y is negative.



```
In [80]: df['x'] > 0.5  ❶
Out[80]: 0     True
         1    False
         2    False
         3    False
         4     True
         5    False
         6    False
         7    False
         8    False
         9    False
         Name: x, dtype: bool

In [81]: (df['x'] > 0) & (df['y'] < 0)  ❷
Out[81]: 0     True
         1    False
         2    False
         3     True
         4     True
         5    False
         6    False
         7    False
         8    False
         9    False
         dtype: bool

In [82]: (df['x'] > 0) | (df['y'] < 0)  ❸
Out[82]: 0     True
         1     True
         2     True
         3     True
         4     True
         5    False
         6    False
         7     True
         8     True
         9     True
         dtype: bool
```

Complex Selection3

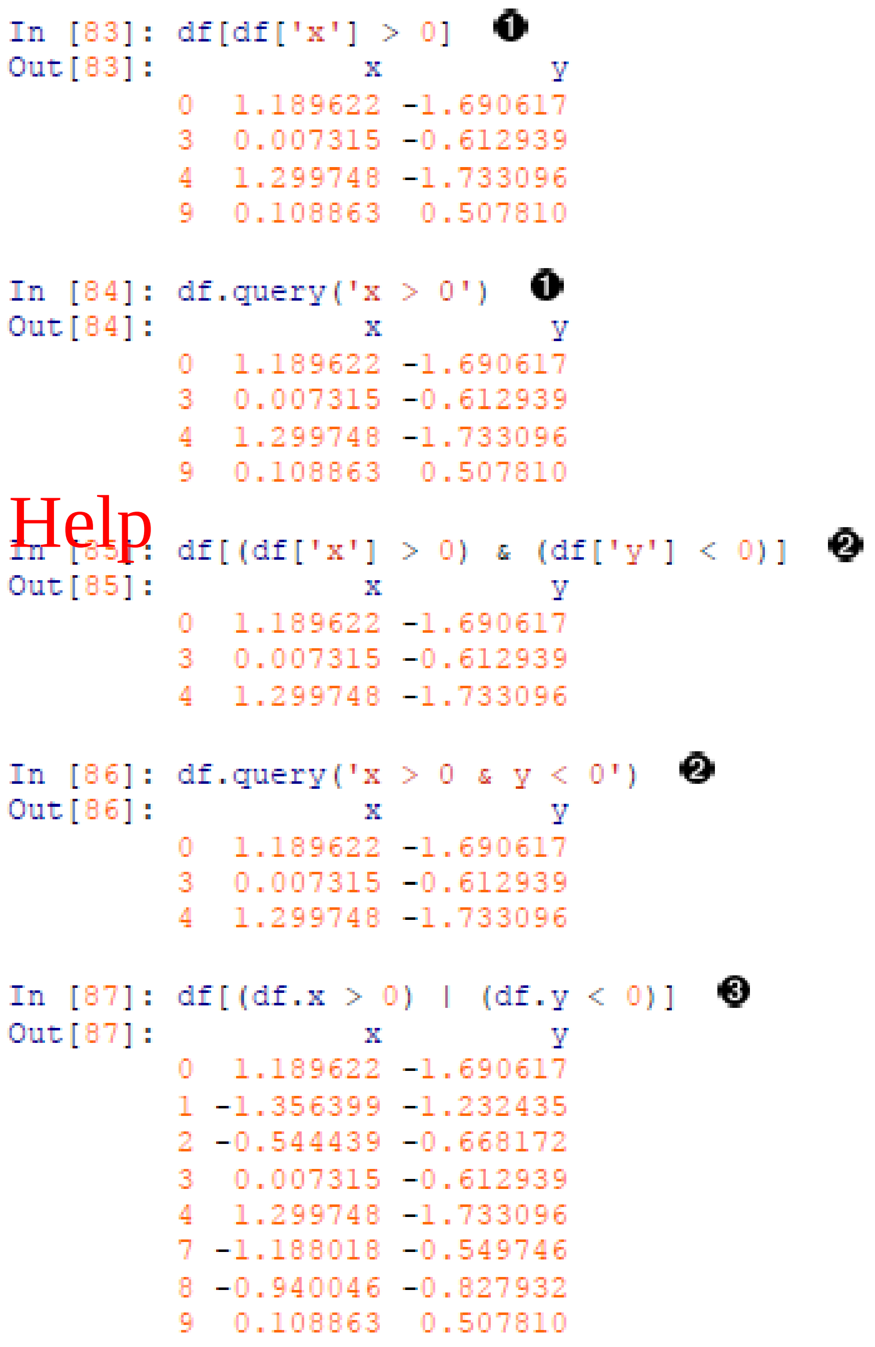
- Using the resulting Boolean `Series` objects, complex data (row) selection is straightforward.
- Alternatively, one can use the `query()` method and pass the conditions as `str` objects:

❶ All rows for which the value in column x is greater than 0.5.

*Note book typo!

❷ All rows for which the value in column x is positive *and* the value in column y is negative.

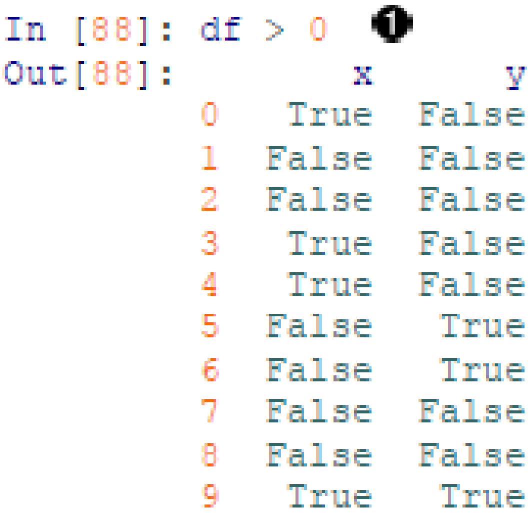
❸ All rows for which the value in column x is positive *or* the value in column y is negative (columns are accessed here via the respective attributes).



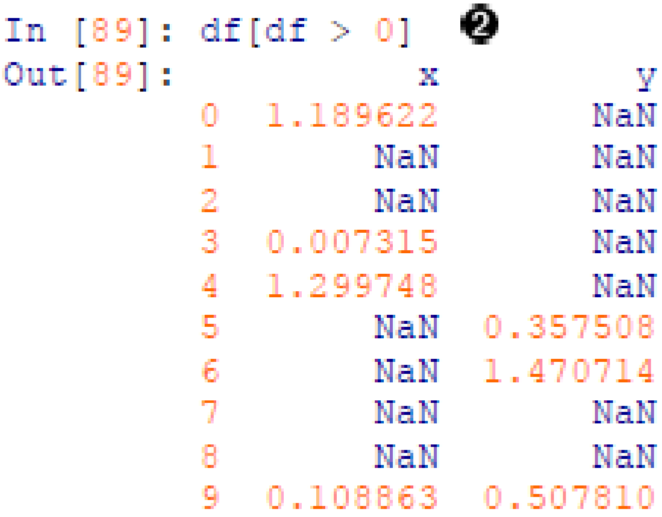
```
In [83]: df[df['x'] > 0]  ❶
Out[83]:
          x         y
0  1.189622 -1.690617
3  0.007315 -0.612939
4  1.299748 -1.733096
9  0.108863  0.507810

In [84]: df.query('x > 0')  ❶
Out[84]:
          x         y
0  1.189622 -1.690617
3  0.007315 -0.612939
4  1.299748 -1.733096
9  0.108863  0.507810

In [85]: df[(df['x'] > 0) & (df['y'] < 0)]  ❷
Out[85]:
          x         y
0  1.189622 -1.690617
3  0.007315 -0.612939
4  1.299748 -1.733096

In [86]: df.query('x > 0 & y < 0')  ❷
Out[86]:
          x         y
0  1.189622 -1.690617
3  0.007315 -0.612939
4  1.299748 -1.733096

In [87]: df[(df.x > 0) | (df.y < 0)]  ❸
Out[87]:
          x         y
0  1.189622 -1.690617
1 -1.356399 -1.232435
2 -0.544439 -0.668172
3  0.007315 -0.612939
4  1.299748 -1.733096
7 -1.188018 -0.549746
8 -0.940046 -0.827932
9  0.108863  0.507810
```

Complex Selection4

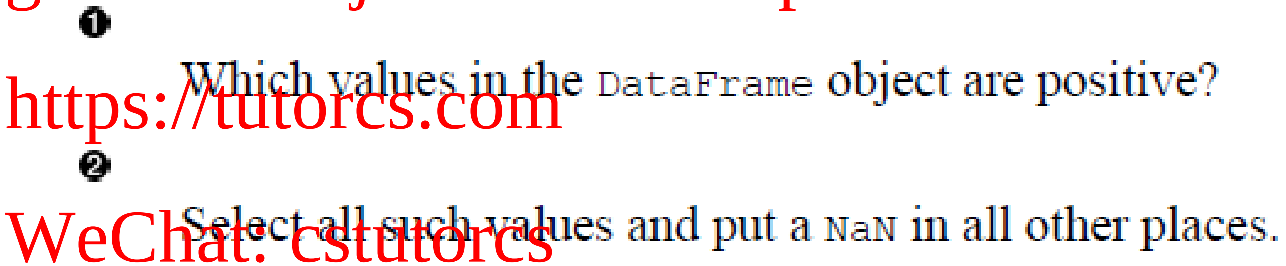
- Comparison operators can also be applied to complete `DataFrame` objects at once:



```
In [88]: df > 0  ❶
Out[88]:
       x      y
0   True  False
1  False  False
2  False  False
3   True  False
4   True  False
5  False   True
6  False   True
7  False  False
8  False  False
9   True   True
```



```
In [89]: df[df > 0]  ❷
Out[89]:
          x         y
0  1.189622       NaN
1       NaN       NaN
2       NaN       NaN
3  0.007315       NaN
4  1.299748       NaN
5       NaN  0.357508
6       NaN  1.470714
7       NaN       NaN
8       NaN       NaN
9  0.108863  0.507810
```

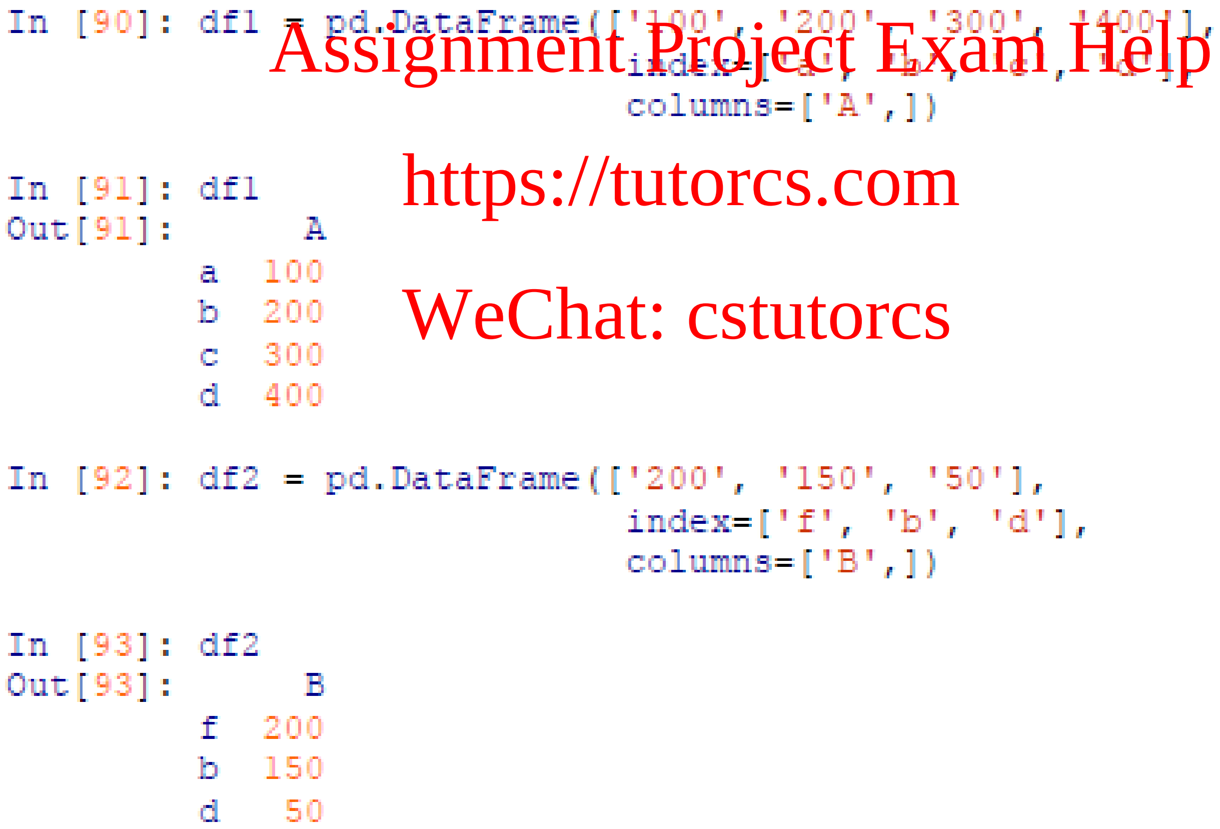


❶ Which values in the `DataFrame` object are positive?

❷ Select all such values and put a `NaN` in all other places.

Concatenation, Joining and Merging

- This section walks through different approaches to combine two simple data sets in the form of `DataFrame` objects. The two simple data sets are:



```
In [90]: df1 = pd.DataFrame(['100', '200', '300', '400'],
                             index=['a', 'b', 'c', 'd'],
                             columns=['A',])

In [91]: df1
Out[91]:      A
         a  100
         b  200
         c  300
         d  400

In [92]: df2 = pd.DataFrame(['200', '150', '50'],
                             index=['f', 'b', 'd'],
                             columns=['B',])

In [93]: df2
Out[93]:      B
         f  200
         b  150
         d   50
```

Concatenation

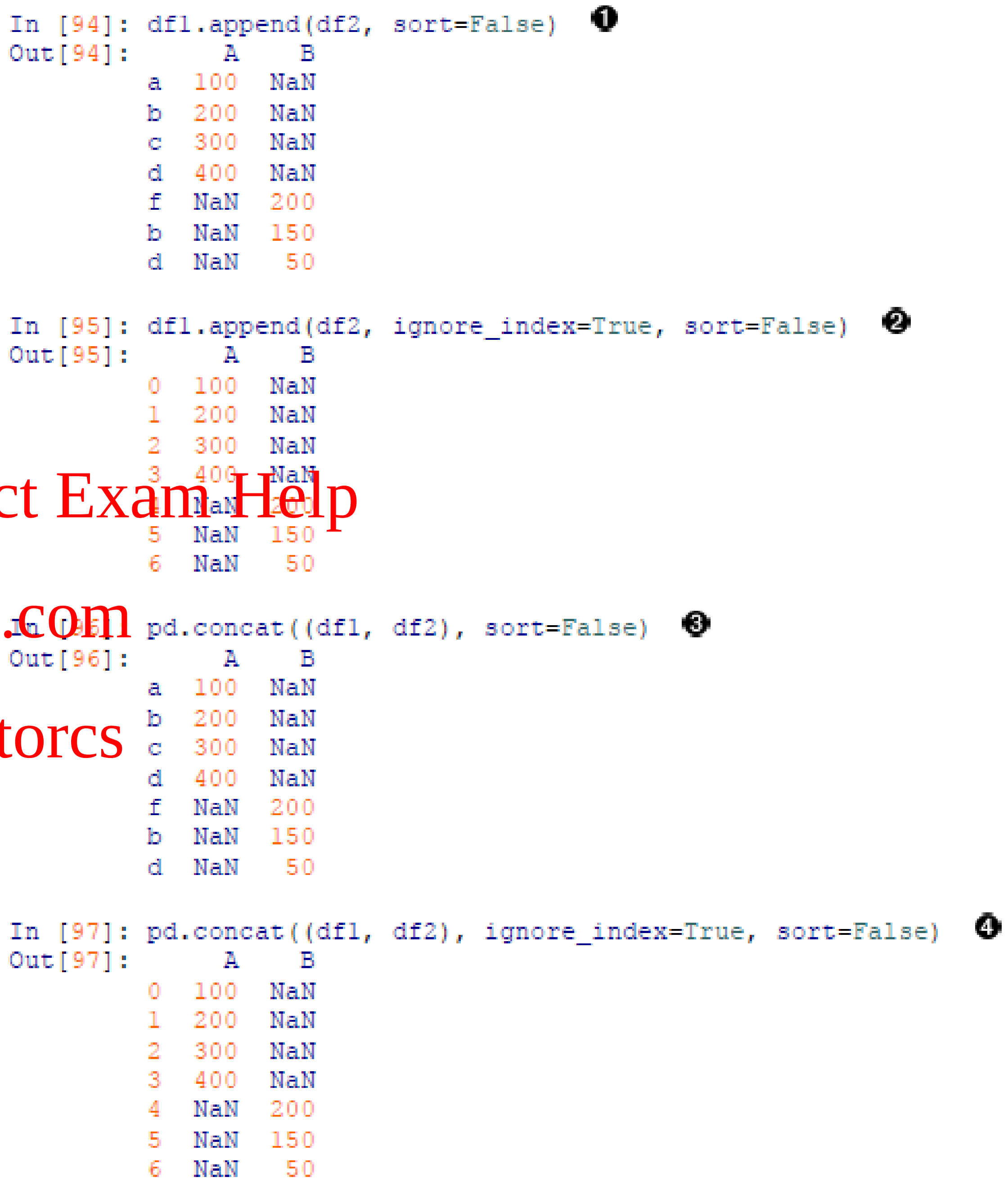
- *Concatenation* or *appending* basically means that rows are added from one `DataFrame` object to another one.
- This can be accomplished via the `append()` method or via the `pd.concat()` function.
- A major consideration is how the index values are handled:

❶ Appends data from `df2` to `df1` as new rows.

❷ Does the same but ignores the indices.

❸ Has the same effect as the first append operation.

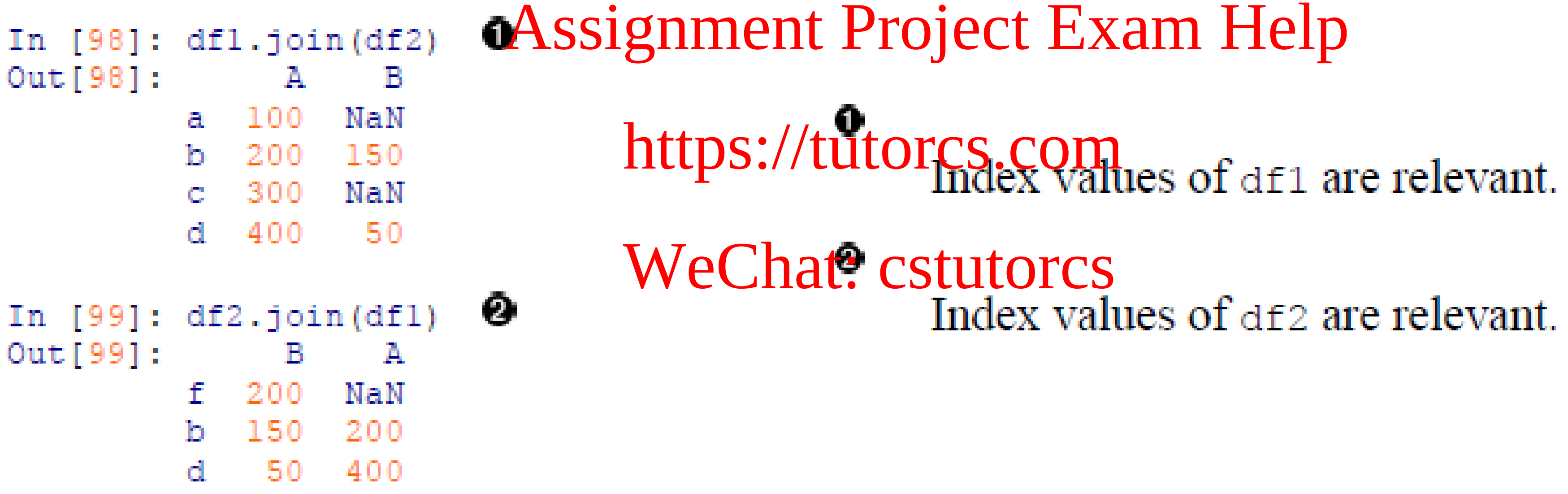
❹ Has the same effect as the second append operation.



```
In [94]: df1.append(df2, sort=False)  ❶
Out[94]:      A    B
         a  100  NaN
         b  200  NaN
         c  300  NaN
         d  400  NaN
         f  NaN  200
         b  NaN  150
         d  NaN   50

In [95]: df1.append(df2, ignore_index=True, sort=False)  ❷
Out[95]:      A    B
         0  100  NaN
         1  200  NaN
         2  300  NaN
         3  400  NaN
         4  NaN  200
         5  NaN  150
         6  NaN   50

In [96]: pd.concat((df1, df2), sort=False)  ❸
Out[96]:      A    B
         a  100  NaN
         b  200  NaN
         c  300  NaN
         d  400  NaN
         f  NaN  200
         b  NaN  150
         d  NaN   50

In [97]: pd.concat((df1, df2), ignore_index=True, sort=False)  ❹
Out[97]:      A    B
         0  100  NaN
         1  200  NaN
         2  300  NaN
         3  400  NaN
         4  NaN  200
         5  NaN  150
         6  NaN   50
```

Joining1

- When joining the two data sets, the sequence of the `DataFrame` objects also matters but in a different way.
- Only the index values from the first `DataFrame` object are used.
- This default behaviour is called a *left join*:



```
In [98]: df1.join(df2) ❶
Out[98]:      A    B
         a  100  NaN
         b  200  150
         c  300  NaN
         d  400   50

In [99]: df2.join(df1) ❷
Out[99]:      B    A
         f  200  NaN
         b  150  200
         d   50  400
```

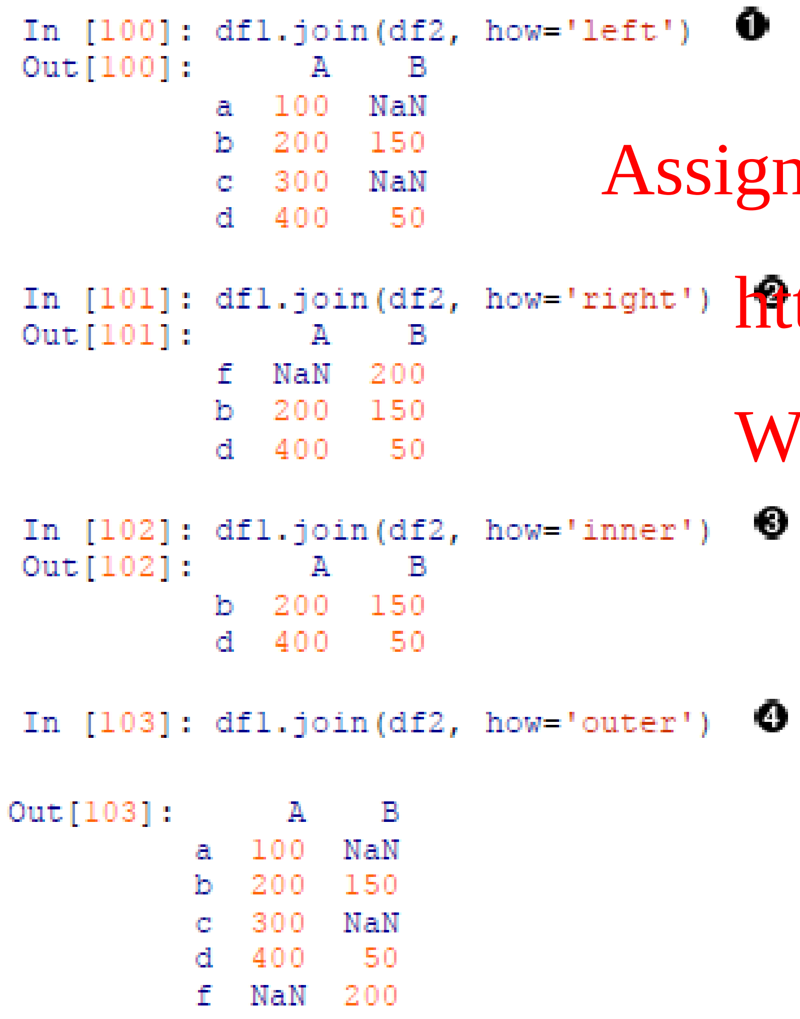
❶ Index values of `df1` are relevant.

❷ Index values of `df2` are relevant.

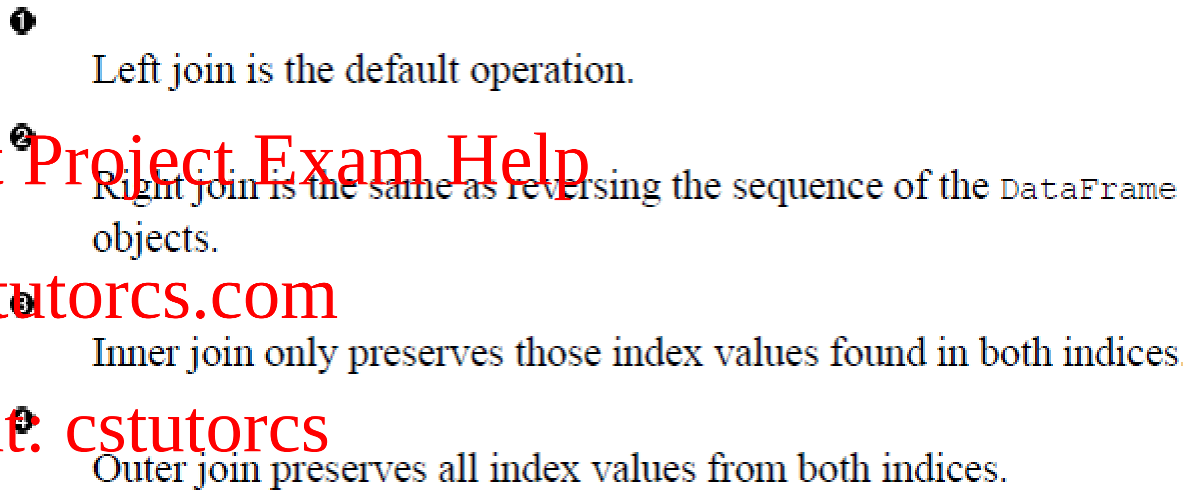
Joining2

Documentation:
https://pandas.pydata.org/docs/reference/api/pandas.DataFrame.join.html?highlight=join#pandas.DataFrame.join

- There are a total of four different join methods available, each leading to a different behaviour with regard to how index values and the corresponding data rows are handled:



```
In [100]: df1.join(df2, how='left')  ❶
Out[100]:
             A    B
        a  100  NaN
        b  200  150
        c  300  NaN
        d  400   50

In [101]: df1.join(df2, how='right')  ❷
Out[101]:
             A    B
        f  NaN  200
        b  200  150
        d  400   50

In [102]: df1.join(df2, how='inner')  ❸
Out[102]:
             A    B
        b  200  150
        d  400   50

In [103]: df1.join(df2, how='outer')  ❹

Out[103]:
             A    B
        a  100  NaN
        b  200  150
        c  300  NaN
        d  400   50
        f  NaN  200
```

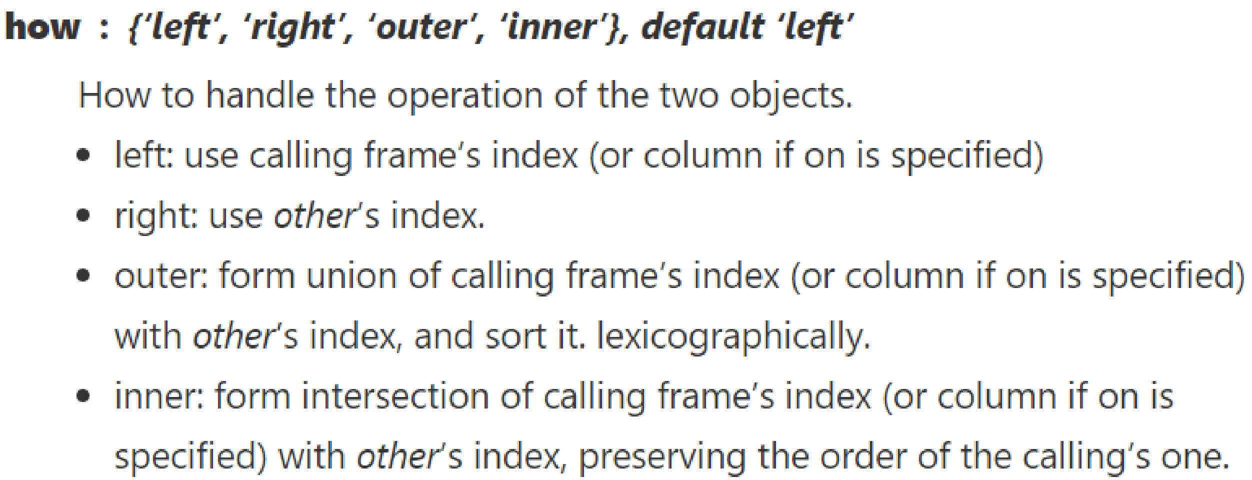


❶ Left join is the default operation.

❷ Right join is the same as reversing the sequence of the `DataFrame` objects.

❸ Inner join only preserves those index values found in both indices.

❹ Outer join preserves all index values from both indices.



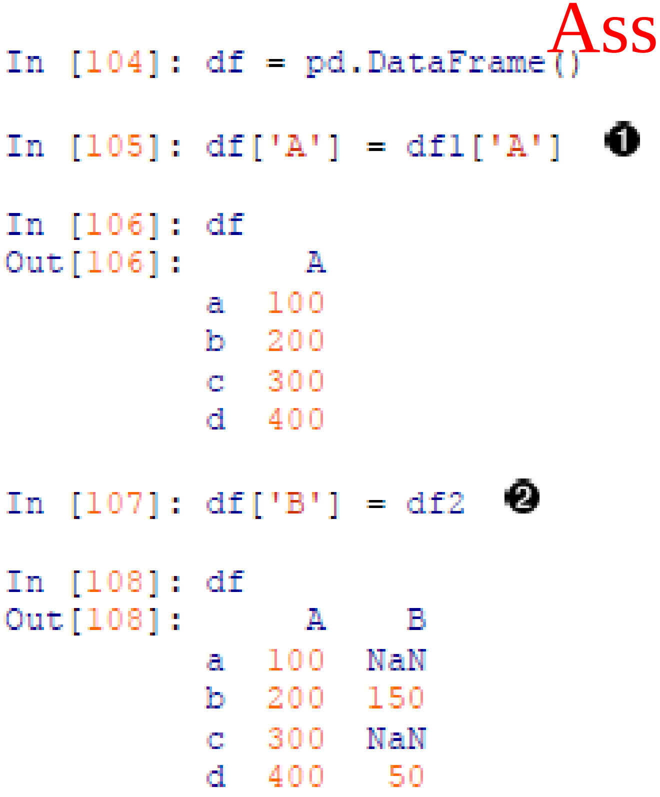
how : *{'left', 'right', 'outer', 'inner'}, default 'left'*

How to handle the operation of the two objects.

- left: use calling frame's index (or column if on is specified)
- right: use *other*'s index.
- outer: form union of calling frame's index (or column if on is specified) with *other*'s index, and sort it. lexicographically.
- inner: form intersection of calling frame's index (or column if on is specified) with *other*'s index, preserving the order of the calling's one.

Joining3

- A join can also happen based on an empty `DataFrame` object.
- In this case, the columns are created *sequentially*, leading to behaviour similar to a left join:



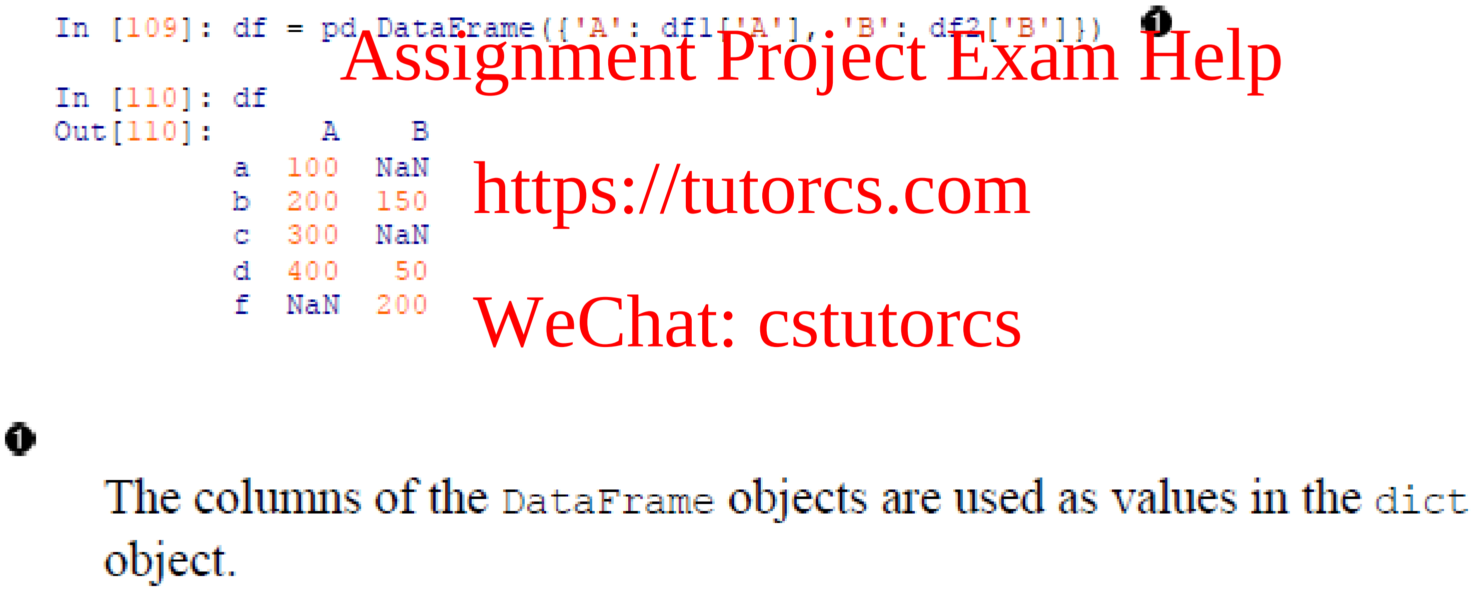
```
In [104]: df = pd.DataFrame()

In [105]: df['A'] = df1['A']  ❶

In [106]: df
Out[106]:
             A
          a  100
          b  200
          c  300
          d  400

In [107]: df['B'] = df2  ❷

In [108]: df
Out[108]:
             A    B
          a  100  NaN
          b  200  150
          c  300  NaN
          d  400   50
```

❶ df1 as first column A.

❷ df2 as second column B.

Joining4

- Making use of a dictionary to combine the data sets yields a result similar to an outer join since the columns are created *simultaneously*:

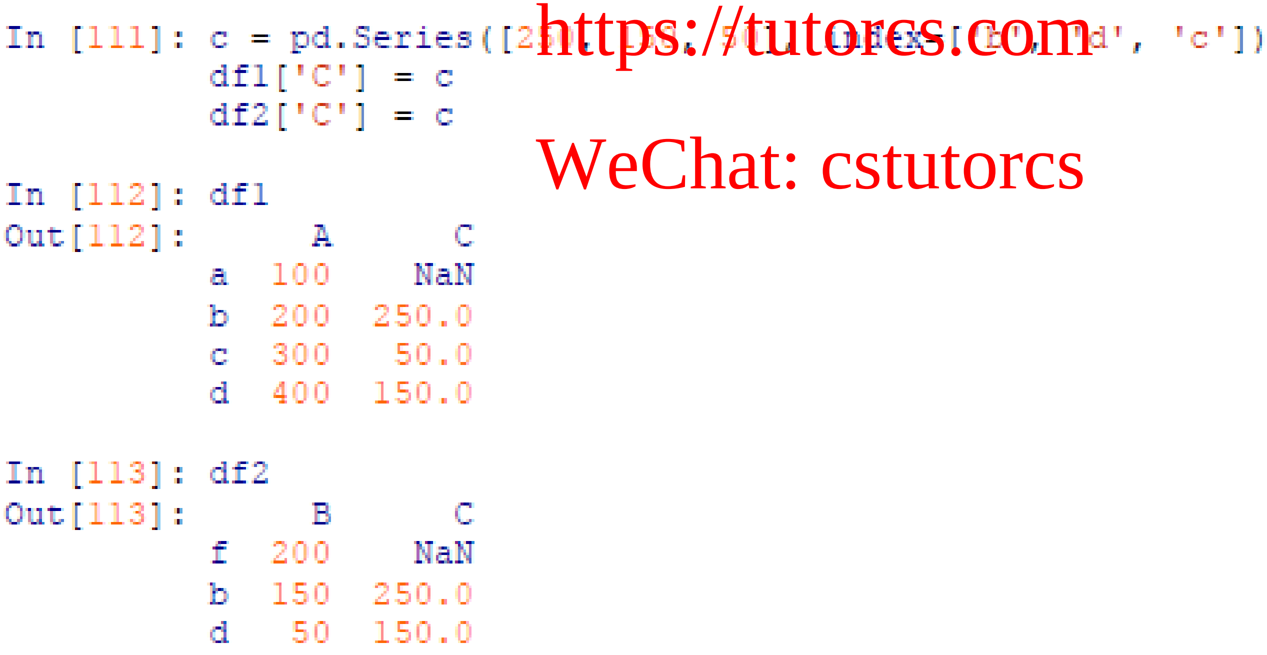


```
In [109]: df = pd.DataFrame({'A': df1['A'], 'B': df2['B']})  ❶

In [110]: df
Out[110]:       A    B
          a   100  NaN
          b   200  150
          c   300  NaN
          d   400   50
          f   NaN  200
```

❶ The columns of the `DataFrame` objects are used as values in the `dict` object.

Merging1

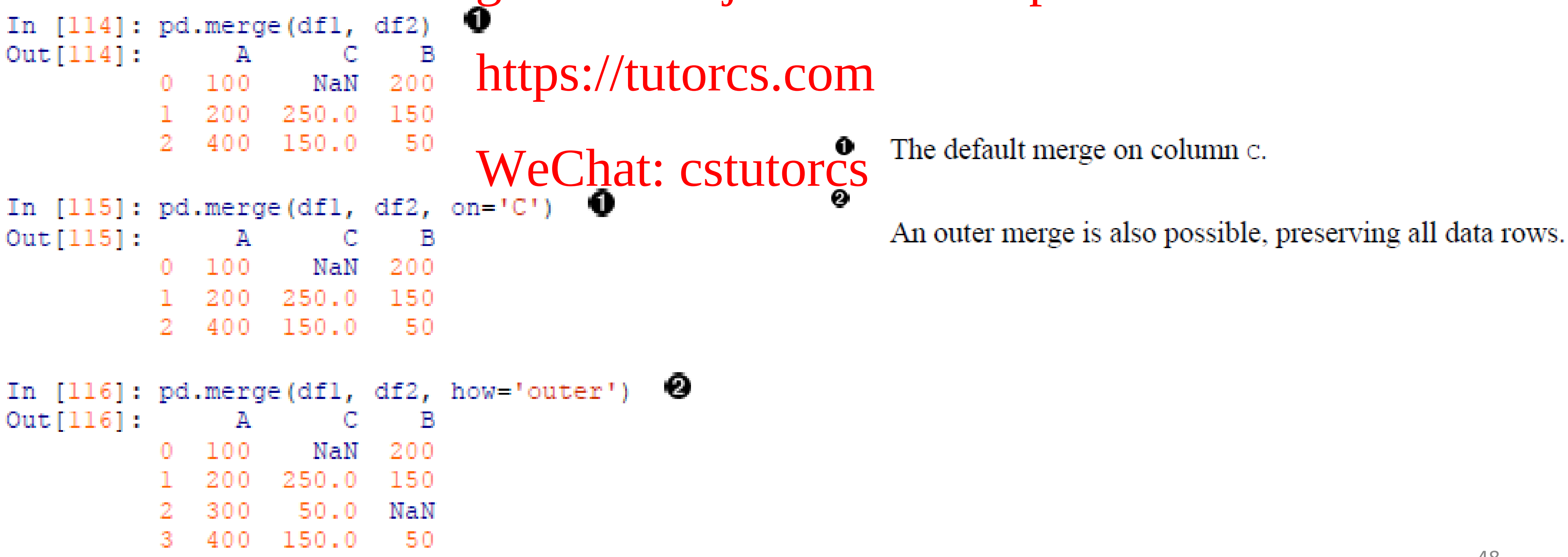
- While a join operation takes place based on the indices of the `DataFrame` objects to be joined, a merge operation typically takes place on a column shared between the two data sets.
- To this end, a new column `C` is added to both original `DataFrame` objects:



```
In [111]: c = pd.Series([250, 150, 50], index=['b', 'd', 'c'])
          df1['C'] = c
          df2['C'] = c

In [112]: df1
Out[112]:      A      C
          a  100    NaN
          b  200  250.0
          c  300   50.0
          d  400  150.0

In [113]: df2
Out[113]:      B      C
          f  200    NaN
          b  150  250.0
          d   50  150.0
```

Merging2

- By default, the merge operation in this case takes place based on the single shared column `C`.
- Other options are available, however, such as an *outer* merge:



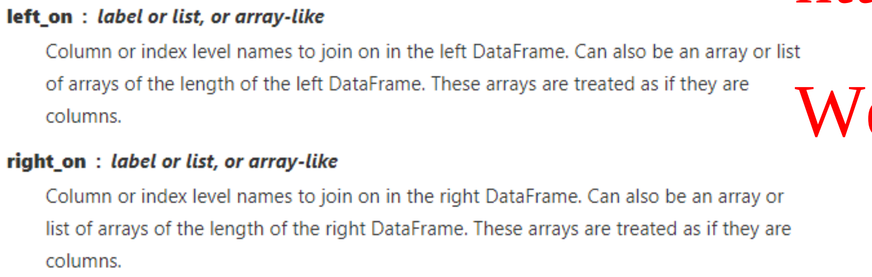
```
In [114]: pd.merge(df1, df2)  ❶
Out[114]:
            A      C    B
         0  100    NaN  200
         1  200  250.0  150
         2  400  150.0   50

In [115]: pd.merge(df1, df2, on='C')  ❶
Out[115]:
            A      C    B
         0  100    NaN  200
         1  200  250.0  150
         2  400  150.0   50

In [116]: pd.merge(df1, df2, how='outer')  ❷
Out[116]:
            A      C    B
         0  100    NaN  200
         1  200  250.0  150
         2  300   50.0  NaN
         3  400  150.0   50
```

❶ The default merge on column `C`.

❷ An outer merge is also possible, preserving all data rows.

Merging3

- Many more types of merge operations are available, a few of which are illustrated in the following code:

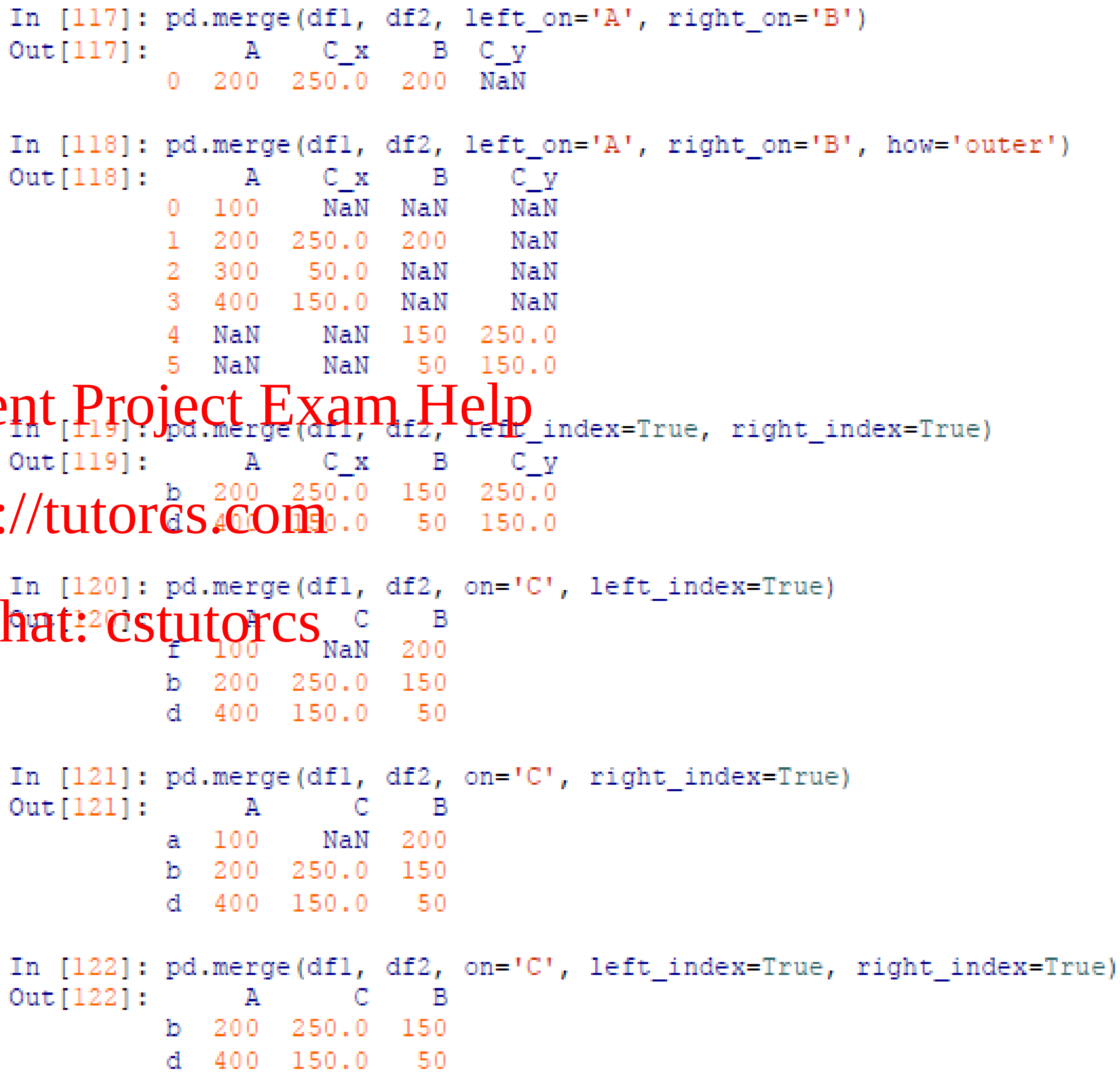


left_on : ***label or list, or array-like***

Column or index level names to join on in the left DataFrame. Can also be an array or list of arrays of the length of the left DataFrame. These arrays are treated as if they are columns.

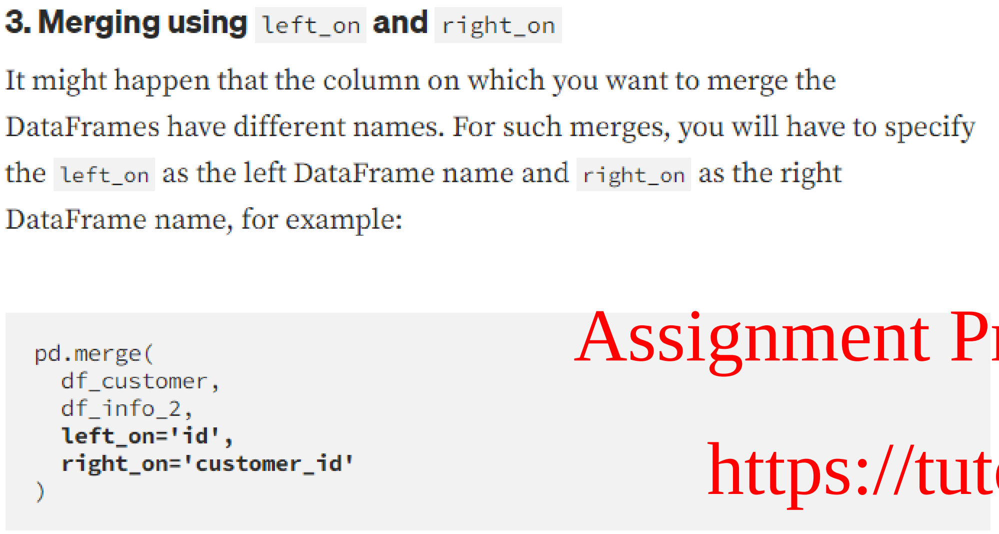
right_on : ***label or list, or array-like***

Column or index level names to join on in the right DataFrame. Can also be an array or list of arrays of the length of the right DataFrame. These arrays are treated as if they are columns.



```
In [117]: pd.merge(df1, df2, left_on='A', right_on='B')
Out[117]:      A    C_x    B  C_y
          0  200  250.0  200  NaN

In [118]: pd.merge(df1, df2, left_on='A', right_on='B', how='outer')
Out[118]:      A    C_x    B    C_y
          0  100    NaN  NaN    NaN
          1  200  250.0  200    NaN
          2  300   50.0  NaN    NaN
          3  400  150.0  NaN    NaN
          4  NaN    NaN  150  250.0
          5  NaN    NaN   50  150.0

In [119]: pd.merge(df1, df2, left_index=True, right_index=True)
Out[119]:      A    C_x    B    C_y
          b  200  250.0  150  250.0
          d  400  150.0   50  150.0

In [120]: pd.merge(df1, df2, on='C', left_index=True)
Out[120]:      A      C    B
          a  100    NaN  200
          b  200  250.0  150
          d  400  150.0   50

In [121]: pd.merge(df1, df2, on='C', right_index=True)
Out[121]:      A      C    B
          a  100    NaN  200
          b  200  250.0  150
          d  400  150.0   50

In [122]: pd.merge(df1, df2, on='C', left_index=True, right_index=True)
Out[122]:      A      C    B
          b  200  250.0  150
          d  400  150.0   50
```

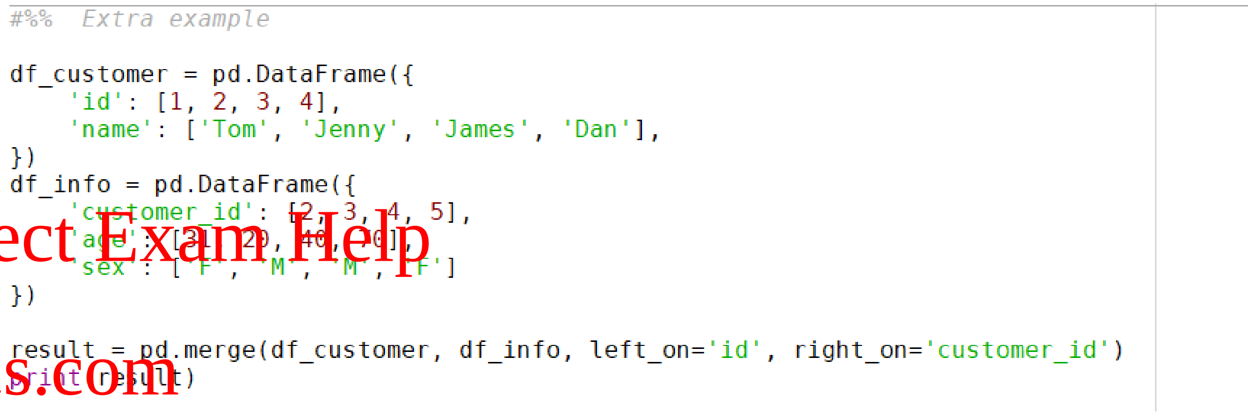
* Merging Extra Example



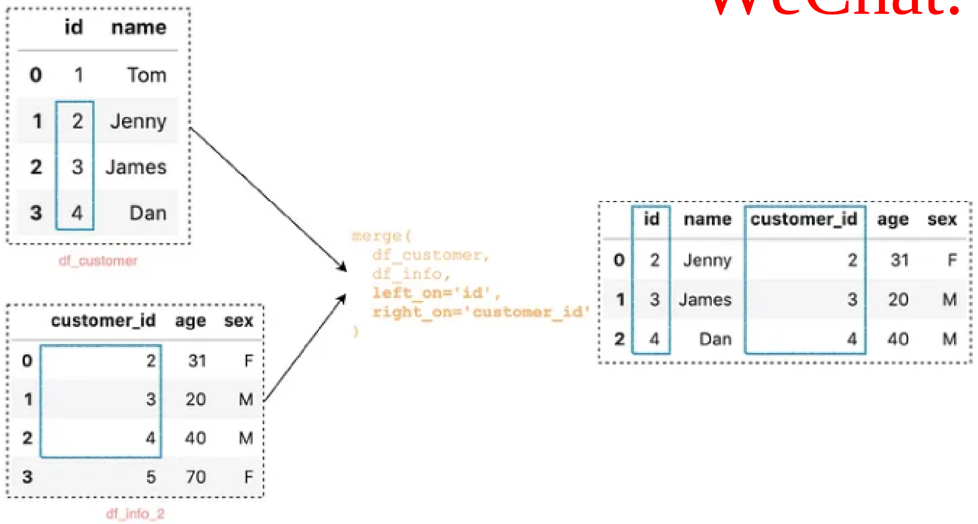
3. Merging using `left_on` and `right_on`

It might happen that the column on which you want to merge the DataFrames have different names. For such merges, you will have to specify the `left_on` as the left DataFrame name and `right_on` as the right DataFrame name, for example:

```
pd.merge(
  df_customer,
  df_info_2,
  left_on='id',
  right_on='customer_id'
)
```



```
#%%  Extra example

df_customer = pd.DataFrame({
    'id': [1, 2, 3, 4],
    'name': ['Tom', 'Jenny', 'James', 'Dan'],
})
df_info = pd.DataFrame({
    'customer_id': [2, 3, 4, 5],
    'age': [31, 20, 40, 70],
    'sex': ['F', 'M', 'M', 'F']
})

result = pd.merge(df_customer, df_info, left_on='id', right_on='customer_id')
print(result)
```



	id	name
0	1	Tom
1	2	Jenny
2	3	James
3	4	Dan

df_customer

	customer_id	age	sex
0	2	31	F
1	3	20	M
2	4	40	M
3	5	70	F

df_info_2

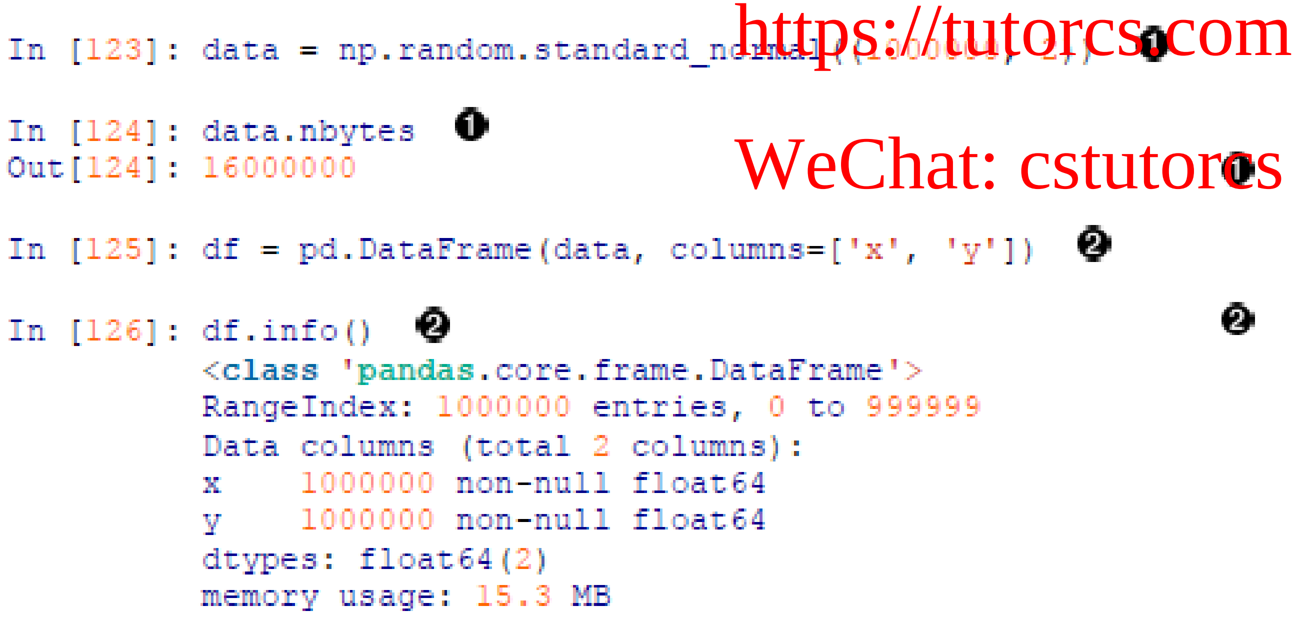
```
merge(
  df_customer,
  df_info,
  left_on='id',
  right_on='customer_id'
)
```

	id	name	customer_id	age	sex
0	2	Jenny	2	31	F
1	3	James	3	20	M
2	4	Dan	4	40	M

https://towardsdatascience.com/all-the-pandas-merge-you-should-know-for-combining-datasets-526b9ecaf184

Performance Aspects1

- Many examples in this chapter illustrate that there are often multiple options to achieve the same goal with `pandas`.
- This section compares such options for adding up two columns element-wise.
- First, the data set, generated with `NumPy`:



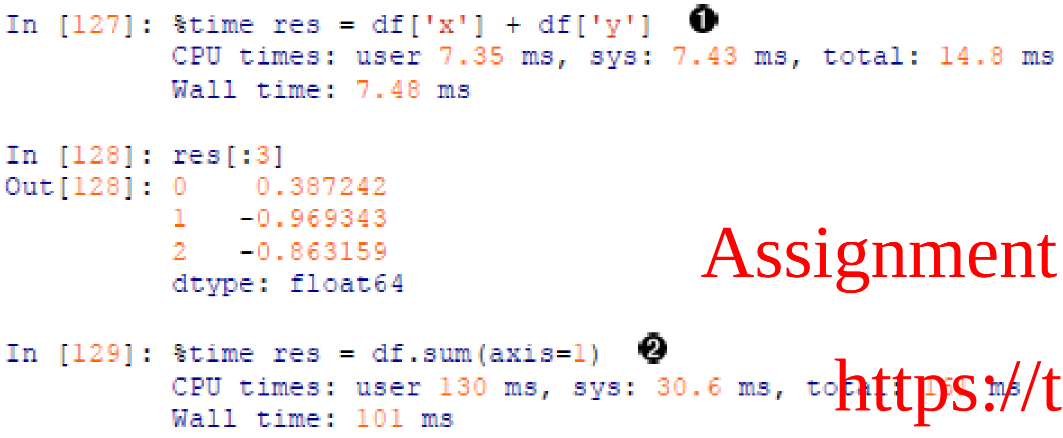
```
In [123]: data = np.random.standard_normal((1000000, 2))  ❶

In [124]: data.nbytes  ❶
Out[124]: 16000000

In [125]: df = pd.DataFrame(data, columns=['x', 'y'])  ❷

In [126]: df.info()  ❷
          <class 'pandas.core.frame.DataFrame'>
          RangeIndex: 1000000 entries, 0 to 999999
          Data columns (total 2 columns):
          x    1000000 non-null float64
          y    1000000 non-null float64
          dtypes: float64(2)
          memory usage: 15.3 MB
```

❶ The `ndarray` object with random numbers.

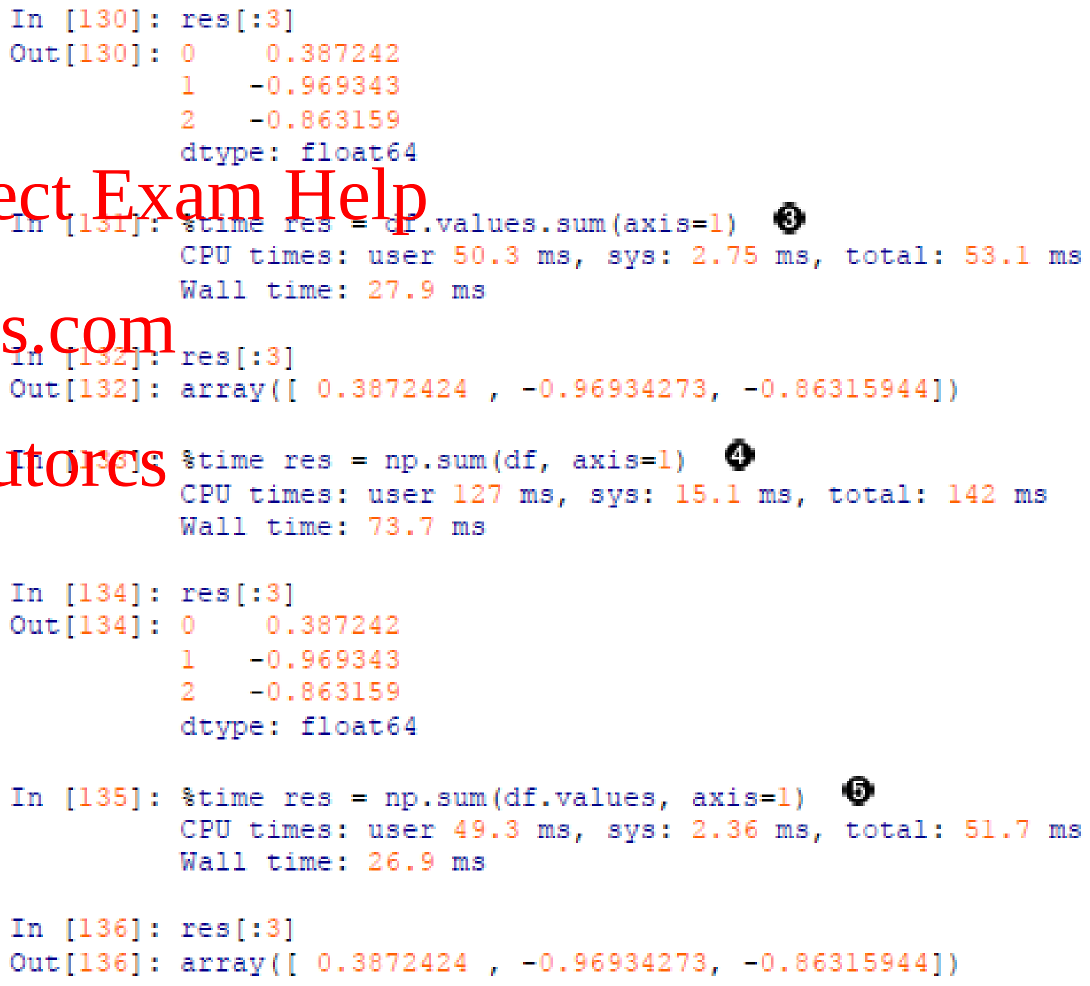
❷ The `DataFrame` object with the random numbers.

Performance Aspects2

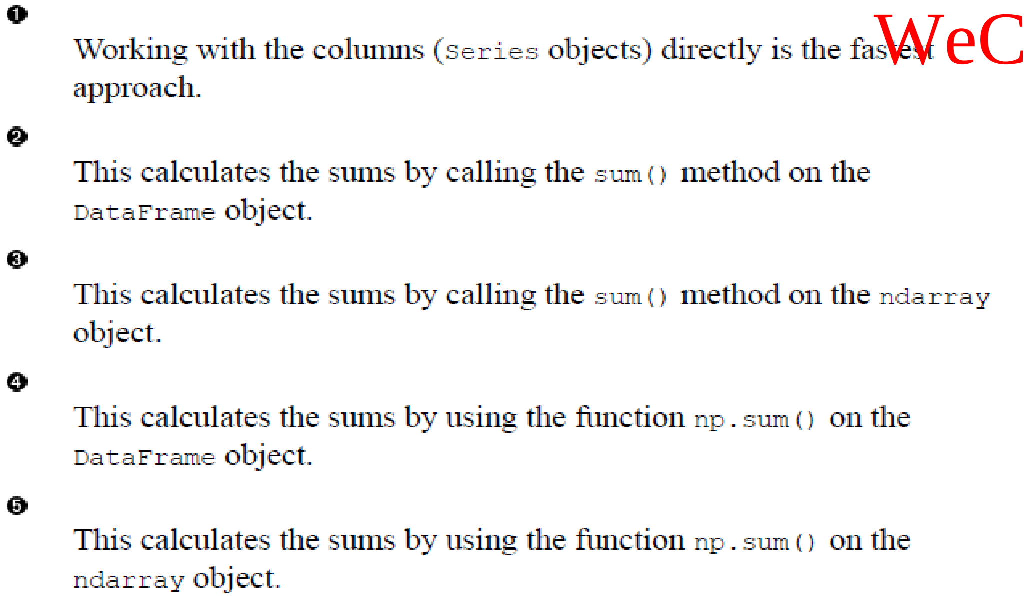
- Second, some options to accomplish the task at hand with performance values:



```
In [127]: %time res = df['x'] + df['y']  ❶
          CPU times: user 7.35 ms, sys: 7.43 ms, total: 14.8 ms
          Wall time: 7.48 ms

In [128]: res[:3]
Out[128]: 0    0.387242
          1   -0.969343
          2   -0.863159
          dtype: float64

In [129]: %time res = df.sum(axis=1)  ❷
          CPU times: user 130 ms, sys: 30.6 ms, total: 161 ms
          Wall time: 101 ms
```



```
In [130]: res[:3]
Out[130]: 0    0.387242
          1   -0.969343
          2   -0.863159
          dtype: float64

In [131]: %time res = df.values.sum(axis=1)  ❸
          CPU times: user 50.3 ms, sys: 2.75 ms, total: 53.1 ms
          Wall time: 27.9 ms

In [132]: res[:3]
Out[132]: array([ 0.3872424 , -0.96934273, -0.86315944])

In [133]: %time res = np.sum(df, axis=1)  ❹
          CPU times: user 127 ms, sys: 15.1 ms, total: 142 ms
          Wall time: 73.7 ms

In [134]: res[:3]
Out[134]: 0    0.387242
          1   -0.969343
          2   -0.863159
          dtype: float64

In [135]: %time res = np.sum(df.values, axis=1)  ❺
          CPU times: user 49.3 ms, sys: 2.36 ms, total: 51.7 ms
          Wall time: 26.9 ms

In [136]: res[:3]
Out[136]: array([ 0.3872424 , -0.96934273, -0.86315944])
```



❶ Working with the columns (`Series` objects) directly is the fastest approach.

❷ This calculates the sums by calling the `sum()` method on the `DataFrame` object.

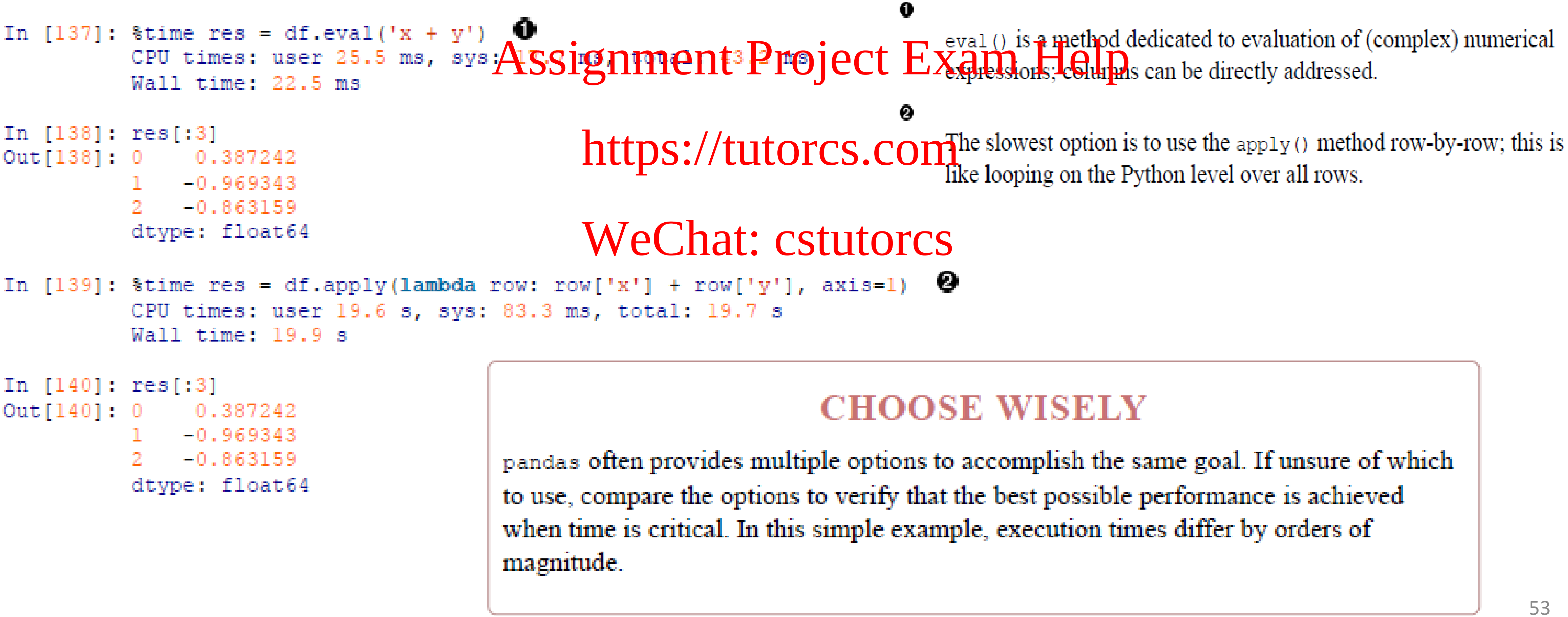
❸ This calculates the sums by calling the `sum()` method on the `ndarray` object.

❹ This calculates the sums by using the function `np.sum()` on the `DataFrame` object.

❺ This calculates the sums by using the function `np.sum()` on the `ndarray` object.

Performance Aspects3

- Finally, two more options which are based on the methods `eval()` and `apply()`, respectively:1



```
In [137]: %time res = df.eval('x + y') ❶
          CPU times: user 25.5 ms, sys: 17.7 ms, total: 43.2 ms
          Wall time: 22.5 ms

In [138]: res[:3]
Out[138]: 0    0.387242
          1   -0.969343
          2   -0.863159
          dtype: float64

In [139]: %time res = df.apply(lambda row: row['x'] + row['y'], axis=1) ❷
          CPU times: user 19.6 s, sys: 83.3 ms, total: 19.7 s
          Wall time: 19.9 s

In [140]: res[:3]
Out[140]: 0    0.387242
          1   -0.969343
          2   -0.863159
          dtype: float64
```

❶ `eval()` is a method dedicated to evaluation of (complex) numerical expressions; columns can be directly addressed.

❷ The slowest option is to use the `apply()` method row-by-row; this is like looping on the Python level over all rows.

CHOOSE WISELY

`pandas` often provides multiple options to accomplish the same goal. If unsure of which to use, compare the options to verify that the best possible performance is achieved when time is critical. In this simple example, execution times differ by orders of magnitude.

Conclusion

- pandas is a powerful tool for data analysis and has become the central package in the so-called *PyData* stack.
- Its DataFrame class is particularly suited to working with tabular data of any kind.
- Most operations on such objects are vectorized, leading not only — as in the NumPy case — to concise code but also to high performance in general.
- In addition, pandas makes working with incomplete data sets convenient (which is not the case with NumPy, for instance).
- pandas and the DataFrame class will be central in many later chapters of the book, where additional features will be used and introduced when necessary.
- **Please also consult Wes McKinney – Python for Data Analysis 2018 2nd Edition Chapter 8 Data Wrangling, Join and Reshape & Chapter 10 Data Aggregation and Group Operations for further knowledge acquisition on these subjects.**