5QQMN534: Algorithmic Finance

Week7: Input / Output Operations (Big Data Analytics)

Yves Hilpisch - Python for Finance 2nd Edition 2019: Chapter 9

Agenda

- Basic I/O with Python
  - Writing Objects to Disks
  - Reading and Writing Text Files
  - Working with SQL Databases
  - Writing and Reading NumPy Arrays
- I/O with Pandas
  - Working with SQL Databases
  - From SQL to pandas
  - Working with Csv Files
  - Working with Excel Files
- I/O with PyTables
  - Working with Tables
  - Working with Compressed Tables
  - Working with Arrays
  - Out of Memory Computations
- I/O with TsTables
  - Sample Data
  - Data Storage
  - Data Retrieval
- Conclusion

Input/Output Operations1

- As a general rule, the majority of data, be it in a finance context or any other application area, is stored on hard disk drives (HDDs) or some other form of permanent storage device, like solid state disks (SSDs) or hybrid disk drives.
- Storage capacities have been steadily increasing over the years, while costs per storage unit (e.g., per megabyte) have been steadily falling.
- At the same time, stored data volumes have been increasing at a much faster pace than the typical random access memory (RAM) available even in the largest machines. This makes it necessary not only to store data to disk for permanent storage, but also to compensate for lack of sufficient RAM by swapping data from RAM to disk and back.
- Input/output (I/O) operations are therefore important tasks when it comes to finance applications and data-intensive applications in general.
- Often they represent the bottleneck for performance-critical computations, since I/O operations cannot typically shuffle data fast enough to the RAM** and from the RAM to the disk. In a sense, CPUs are often “starving” due to slow I/O operations.

** Here, no distinction is made between different levels of RAM and processor caches. The optimal use of current memory architectures is a topic in itself.

Input/Output Operations2

- Although the majority of today's financial and corporate analytics efforts are confronted with big data (e.g., of petascale size), single analytics tasks generally use data subsets that fall in the "mid" data category. A study by Microsoft Research concludes:
- Our measurements as well as other recent work shows that the majority of real-world analytic jobs process less than 100 GB of input, but popular infrastructures such as Hadoop/MapReduce were originally designed for petascale processing. Appuswamy et al. (2013)
- In terms of frequency, single financial analytics tasks generally process data of not more than a couple of gigabytes (GB) in size — and this is a sweet spot for Python and the libraries of its scientific stack, such as `NumPy`, `pandas`, and `PyTables`. Data sets of such a size can also be analyzed in memory, leading to generally high speeds with today's CPUs and GPUs.
- However, the data has to be read into RAM and the results have to be written to disk, meanwhile ensuring that today's performance requirements are met.

Data Measurement Chart	
Data Measurement	**Size**
Bit	Single Binary Digit (1 or 0)
Byte	8 bits
Kilobyte (KB)	1,024 Bytes
Megabyte (MB)	1,024 Kilobytes
Gigabyte (GB)	1,024 Megabytes
Terabyte (TB)	1,024 Gigabytes
Petabyte (PB)	1,024 Terabytes
Exabyte (EB)	1,024 Petabytes

Input/Output Operations3

- This chapter addresses the following topics:

"Basic I/O with Python"

Python has built-in functions to serialize and store any object on disk and to read it from disk into RAM; apart from that, Python is strong when it comes to working with text files and SQL databases. `NumPy` also provides dedicated functions for fast binary storage and retrieval of `ndarray` objects.

"I/O with pandas"

The `pandas` library provides a plenitude of convenience functions and methods to read data stored in different formats (e.g., CSV, JSON) and to write data to files in diverse formats.

"I/O with PyTables"

`PyTables` uses the HDF5 standard with hierarchical database structure and binary storage to accomplish fast I/O operations for large data sets; speed often is only bound by the hardware used.

"I/O with TsTables"

`TsTables` is a package that builds on top of `PyTables` and allows for fast storage and retrieval of time series data.

Basic I/O with Python

- Python itself comes with a multitude of I/O capabilities, some optimized for performance, others more for flexibility.
- In general, however, they are easily used in interactive as well as in production settings.

Writing Objects to Disks1

- For later use, for documentation, or for sharing with others, one might want to store Python objects on disk.
- One option is to use the `pickle` module.
- This module can serialize the majority of Python objects.
- *Serialization* refers to the conversion of an object (hierarchy) to a byte stream; *deserialization* is the opposite operation.
- As usual, some imports and customizations with regard to plotting first:


```
In [1]: from pylab import plt, mpl
        plt.style.use('seaborn')
        mpl.rcParams['font.family'] = 'serif'
        %matplotlib inline
```

In computing, **serialization** (US spelling) or **serialisation** (UK spelling) is the process of translating a data structure or object state into a format that can be stored (for example, in a file or memory data buffer) or transmitted (for example, over a computer network) and reconstructed later (possibly in a different computer environment)
https://web.archive.org/web/20150405013606/http://isocpp.org/wiki/faq/serialization

Writing Objects to Disks2

- The example that follows works with (pseudo-)random data, this time stored in a `list` object:

```
In [2]: import pickle  ❶
        import numpy as np
        from random import gauss  ❷

In [3]: a = [gauss(1.5, 2) for i in range(1000000)]  ❸

In [4]: path = '/Users/yves/Temp/data/'  ❹

In [5]: pkl_file = open(path + 'data.pkl', 'wb')  ❺
```

❶ Imports the `pickle` module from the standard library.

❷ Import `gauss` to generate normally distributed random numbers.

❸ Creates a larger `list` object with random numbers.

❹ Specifies the path where to store the data files.

❺ Opens a file for writing in binary mode (`wb`).

gauss() is an inbuilt method of the random module. It is used to return a random floating point number with gaussian distribution.

- The "wb" mode opens the file in binary format for writing. Unlike text files, binary files are not human-readable.
- When opened using any text editor, the data is unrecognizable.
- If you write in “wb” mode you must read in “rb”
- https://quick-adviser.com/what-is-binary-mode/#What_is_binary_mode

*Normally, files are opened in text mode, that means, you read and write strings from and to the file, which are encoded in a specific encoding. If encoding is not specified, the default is platform dependent (see `open()`). `'b'` appended to the mode opens the file in binary mode: **now the data is read and written in the form of bytes objects**. This mode should be used for all files that don't contain text.*

Writing Objects to Disks3

- The two major functions to serialize and deserialize Python objects are `pickle.dump()`, for writing objects, and `pickle.load()`, for loading them into memory:

```
In [6]: %time pickle.dump(a, pkl_file) ❶
        CPU times: user 37.2 ms, sys: 15.3 ms, total: 52.5 ms
        Wall time: 50.8 ms

In [7]: pkl_file.close() ❷

In [8]: ll $path* ❸
        -rw-r--r--  1 yves  staff  9002006 Oct 19 12:11
         /Users/yves/Temp/data/data.pkl

In [9]: pkl_file = open(path + 'data.pkl', 'rb') ❹

In [10]: %time b = pickle.load(pkl_file) ❺
         CPU times: user 34.1 ms, sys: 16.7 ms, total: 50.8 ms
         Wall time: 48.7 ms

In [11]: a[:3]
Out[11]: [6.517874180585469, -0.5552400459507827, 2.8488946310833096]

In [12]: b[:3]
Out[12]: [6.517874180585469, -0.5552400459507827, 2.8488946310833096]

In [13]: np.allclose(np.array(a), np.array(b)) ❻
Out[13]: True
```

❶ Serializes the object `a` and saves it to the file.

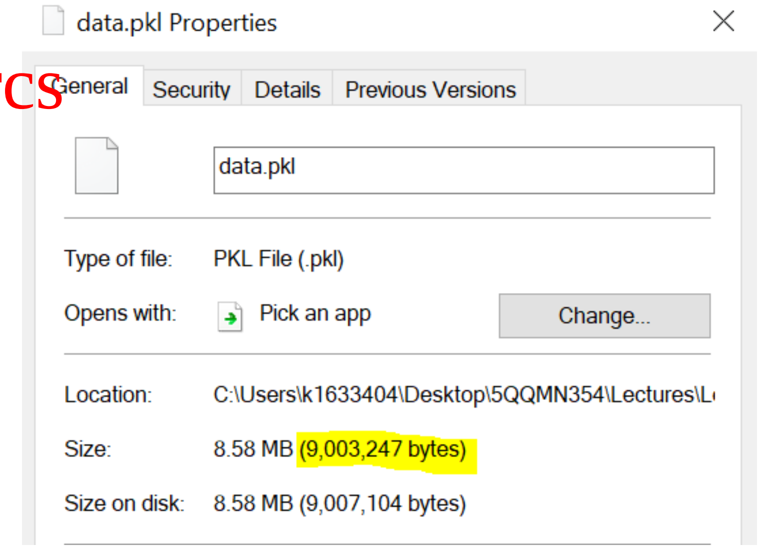
❷ Closes the file.

❸ Shows the file on disk and its size (Mac/Linux).

❹ Opens the file for reading in binary mode (`rb`).

❺ Reads the object from disk and deserializes it.

❻ Converting `a` and `b` to `ndarrary` objects, `np.allclose()` verifies that both contain the same data (numbers).



Note: Python file save size on Windows is this. If you right click properties this matches the yellow highlighted size

9003247

https://docs.python.org/3/library/pickle.html#pickle.dump

https://docs.python.org/3/library/pickle.html#pickle.load

Writing Objects to Disks4

- Storing and retrieving a single object with `pickle` obviously is quite simple.
- What about two objects?

```
In [14]: pkl_file = open(path + 'data.pkl', 'wb')

In [15]: %time pickle.dump(np.array(a), pkl_file)  ❶
         CPU times: user 58.1 ms, sys: 6.09 ms, total: 64.2 ms
         Wall time: 32.5 ms

In [16]: %time pickle.dump(np.array(a) ** 2, pkl_file)  ❷
         CPU times: user 66.7 ms, sys: 7.22 ms, total: 73.9 ms
         Wall time: 39.3 ms

In [17]: pkl_file.close()

In [18]: ll $path*  ❸
         -rw-r--r--  1 yves  staff  16000322 Oct 19 12:11
          /Users/yves/Temp/data/data.pkl
```


❶ Serializes the `ndarray` version of `a` and saves it.

❷ Serializes the squared `ndarray` version of `a` and saves it.

❸ The file now has roughly double the size from before.

Writing Objects to Disks5

- What about reading the two `ndarray` objects back into memory?

```
In [19]: pkl_file = open(path + 'data.pkl', 'rb')

In [20]: x = pickle.load(pkl_file) ❶
         x[:4]
Out[20]: array([ 6.51787418, -0.55524005,  2.84889463,  5.94489175])

In [21]: y = pickle.load(pkl_file) ❷
         y[:4]
Out[21]: array([42.48268383,  0.30829151,  8.11620062, 35.34173791])

In [22]: pkl_file.close()
```


❶ This retrieves the object that was stored *first*.

❷ This retrieves the object that was stored *second*.

Writing Objects to Disks6

Warning: The pickle module is not intended to be secure against erroneous or maliciously constructed data. Never unpickle data received from an untrusted or unauthenticated source.

- Obviously, `pickle` stores objects according to the *first in, first out* (FIFO) principle.
- There is one major problem with this: there is no metainformation available to the user to know beforehand what is stored in a `pickle` file.
- A sometimes helpful workaround is to not store single objects, but a `dict` object containing all the other objects:


```
In [23]: pkl_file = open(path + 'data.pkl', 'wb')
         pickle.dump({'x': x, 'y': y}, pkl_file)  ❶
         pkl_file.close()
```



```
In [24]: pkl_file = open(path + 'data.pkl', 'rb')
         data = pickle.load(pkl_file)  ❷
         pkl_file.close()
         for key in data.keys():
             print(key, data[key][:4])
         x [ 6.51787418 -0.55524005  2.84889463  5.94489175]
         y [42.48268383  0.30829151  8.11620062 35.34173791]
```


❶ Stores a `dict` object containing the two `ndarray` objects.

❷ Retrieves the `dict` object.

- In computing and in systems theory, **FIFO** an acronym for **first in, first out** (the first in is the first out) is a method for organizing the manipulation of a data structure (often, specifically a data buffer) where the oldest (first) entry, or "head" of the queue, is processed first.
- Such processing is analogous to servicing people in a queue area on a first-come, first-served (FCFS) basis, i.e. in the same sequence in which they arrive at the queue's tail
- https://en.wikipedia.org/wiki/FIFO_(computing_and_electronics)

Writing Objects to Disks7

- This approach requires writing and reading all the objects at once, but this is a compromise one can probably live with in many circumstances given the higher convenience it brings along.

COMPATIBILITY ISSUES

The use of `pickle` for the serialization of objects is generally straightforward. However, it might lead to problems when, e.g., a Python package is upgraded and the new version of the package cannot work anymore with the serialized object from the older version. It might also lead to problems when sharing such an object across platforms and operating systems. It is therefore in general advisable to work with the built-in reading and writing capabilities of the packages such as `NumPy` and `pandas` that are discussed in the following sections.

Reading and Writing Text Files1

- Text processing can be considered a strength of Python.
- In fact, many corporate and scientific users use Python for exactly this task.
- With Python one has multiple options to work with `str` objects, as well as with text files in general.
- Assume the case of quite a large set of data that shall be shared as a CSV file.
- Although such files have a special internal structure, they are basically plain text files.
- The following code creates a dummy data set as an `ndarray` object, creates a `DatetimeIndex` object, combines the two, and stores the data as a CSV text file:

Reading and Writing Text Files2

❶ Defines the number of rows for the data set.

❷ Creates the `ndarray` object with the random numbers.

❸ Creates a `DatetimeIndex` object of appropriate length (hourly intervals).

❹ Opens a file for writing (`w`).

❺ Defines the header row (column labels) and writes it as the first line.


```
In [26]: import pandas as pd

In [27]: rows = 5000  ❶
         a = np.random.standard_normal((rows, 5)).round(4)  ❷

In [28]: a  ❷
Out[28]: array([[-0.0892, -1.0508, -0.5942,  0.3367,  1.508 ],
                [ 2.1046,  3.2623,  0.704 , -0.2651,  0.4461],
                [-0.0482, -0.9221,  0.1332,  0.1192,  0.7782],
                ...,
                [ 0.3026, -0.2005, -0.9947,  1.0203, -0.6578],
                [-0.7031, -0.6989, -0.8031, -0.4271,  1.9963],
                [ 2.4573,  2.2151,  0.158 , -0.7039, -1.0337]])

In [29]: t = pd.date_range(start='2019/1/1', periods=rows, freq='H')  ❸

In [30]: t  ❸
Out[30]: DatetimeIndex(['2019-01-01 00:00:00', '2019-01-01 01:00:00',
                        '2019-01-01 02:00:00', '2019-01-01 03:00:00',
                        '2019-01-01 04:00:00', '2019-01-01 05:00:00',
                        '2019-01-01 06:00:00', '2019-01-01 07:00:00',
                        '2019-01-01 08:00:00', '2019-01-01 09:00:00',
                        ...
                        '2019-07-27 22:00:00', '2019-07-27 23:00:00',
                        '2019-07-28 00:00:00', '2019-07-28 01:00:00',
                        '2019-07-28 02:00:00', '2019-07-28 03:00:00',
                        '2019-07-28 04:00:00', '2019-07-28 05:00:00',
                        '2019-07-28 06:00:00', '2019-07-28 07:00:00'],
                       dtype='datetime64[ns]', length=5000, freq='H')

In [31]: csv_file = open(path + 'data.csv', 'w')  ❹

In [32]: header = 'date,no1,no2,no3,no4,no5\n'  ❺

In [33]: csv_file.write(header)  ❺
Out[33]: 25
```

Reading and Writing Text Files3

```
In [34]: for t_, (no1, no2, no3, no4, no5) in zip(t, a):  ❻
             s = '{},{},{},{},{},{}\n'.format(t_, no1, no2, no3, no4, no5)  ❼
             csv_file.write(s)  ❽

In [35]: csv_file.close()
```


❻

Combines the data row-wise ...

❼

... into `str` objects ...

❽

... and writes it line-by-line (appending to the CSV text file).

Reading and Writing Text Files4

- The other way around works quite similarly.
- First, open the now-existing CSV file.
- Second, read its content line-by-line using the `.readline()` or `.readlines()` methods of the `file` object:

```
In [37]: csv_file = open(path + 'data.csv', 'r')  ❶

In [38]: for i in range(5):
             print(csv_file.readline(), end='')  ❷
         date,no1,no2,no3,no4,no5
         2019-01-01 00:00:00,-0.0892,-1.0508,-0.5942,0.3367,1.508
         2019-01-01 01:00:00,2.1046,3.2623,0.704,-0.2651,0.4461

         2019-01-01 02:00:00,-0.0482,-0.9221,0.1332,0.1192,0.7782
         2019-01-01 03:00:00,-0.359,-2.4955,0.6164,0.712,-1.4328

In [39]: csv_file.close()

In [40]: csv_file = open(path + 'data.csv', 'r')  ❶

In [41]: content = csv_file.readlines()  ❸

In [42]: content[:5]  ❹
Out[42]: ['date,no1,no2,no3,no4,no5\n',
          '2019-01-01 00:00:00,-0.0892,-1.0508,-0.5942,0.3367,1.508\n',
          '2019-01-01 01:00:00,2.1046,3.2623,0.704,-0.2651,0.4461\n',
          '2019-01-01 02:00:00,-0.0482,-0.9221,0.1332,0.1192,0.7782\n',
          '2019-01-01 03:00:00,-0.359,-2.4955,0.6164,0.712,-1.4328\n']

In [43]: csv_file.close()
```


❶ Opens the file for reading (`r`).

❷ Reads the file contents line-by-line and prints them.

❸ Reads the file contents in a single step …

❹ … the result of which is a `list` object with all lines as separate `str` objects.

Reading and Writing Text Files5

- CSV files are so important and commonplace that there is a `csv` module in the Python standard library that simplifies the processing of these files.
- Two helpful reader (iterator) objects of the `csv` module return either a `list` of `list` objects or a `list` of `dict` objects:

```
In [44]: import csv

In [45]: with open(path + 'data.csv', 'r') as f:
             csv_reader = csv.reader(f)  ❶
             lines = [line for line in csv_reader]

         ['2019-01-01 03:00:00', '-0.359', '-2.4955', '0.6164', '0.712',
          '-1.4328']]

In [47]: with open(path + 'data.csv', 'r') as f:
             csv_reader = csv.DictReader(f)  ❷
             lines = [line for line in csv_reader]

In [48]: lines[:3]  ❷
Out[48]: [OrderedDict([('date', '2019-01-01 00:00:00'),
                       ('no1', '-0.0892'),
                       ('no2', '-1.0508'),
                       ('no3', '-0.5942'),
                       ('no4', '0.3367'),
                       ('no5', '1.508')]),
          OrderedDict([('date', '2019-01-01 01:00:00'),
                       ('no1', '2.1046'),
                       ('no2', '3.2623'),
                       ('no3', '0.704'),
                       ('no4', '-0.2651'),
                       ('no5', '0.4461')]),
          OrderedDict([('date', '2019-01-01 02:00:00'),
                       ('no1', '-0.0482'),
                       ('no2', '-0.9221'),
                       ('no3', '0.1332'),
                       ('no4', '0.1192'),
                       ('no5', '0.7782')])]
```


❶ `csv.reader()` returns every single line as a `list` object.

❷ `csv.DictReader()` returns every single line as an `OrderedDict`, which is a special case of a `dict` object.

Manual: https://docs.python.org/3/library/csv.html

Working with SQL Databases1

- Python can work with any kind of Structured Query Language (SQL) database, and in general also with any kind of NoSQL database.
- One SQL or *relational* database that is delivered with Python by default is SQLite3.
- With it, the basic Python approach to SQL databases can be easily illustrated:** (see next slide for code)

**

- For an overview of available database connectors for Python, visit *https://wiki.python.org/moin/DatabaseInterfaces*.
- Instead of working directly with relational databases, object relational mappers such as SQLAlchemy often prove useful.
- https://www.sqlalchemy.org/
- They introduce an abstraction layer that allows for more Pythonic, object-oriented code.
- They also allow you to more easily exchange one relational database for another in the backend.
- Basic SQL Tutorials: https://www.w3schools.com/sql/

A relational database is a collection of data items with pre-defined relationships between them. These items are organized as a set of tables with columns and rows. Tables are used to hold information about the objects to be represented in the database. Each column in a table holds a certain kind of data and a field stores the actual value of an attribute. The rows in the table represent a collection of related values of one object or entity. Each row in a table could be marked with a unique identifier called a primary key, and rows among multiple tables can be made related using foreign keys. This data can be accessed in many different ways without reorganizing the database tables themselves.

Working with SQL Databases2

```
# Note all SQL SYNTAX should be CAPITALS
```

❶ Opens a database connection; a file is created if it does not exist.

❷ A SQL query that creates a table with three columns.[3]

❸ Executes the query …

❹ … and commits the changes.

❺ Defines a short alias for the `con.execute()` method.

❻ Fetches metainformation about the database, showing the just-created table as the single object.

```
In [50]: import sqlite3 as sq3

In [51]: con = sq3.connect(path + 'numbs.db')  ❶

In [52]: query = 'CREATE TABLE numbs (Date date, No1 real, No2 real)'  ❷

In [53]: con.execute(query)  ❸
Out[53]: <sqlite3.Cursor at 0x102655f10>

In [54]: con.commit()  ❹

In [55]: q = con.execute  ❺

In [56]: q('SELECT * FROM sqlite_master').fetchall()  ❻
Out[56]: [('table',
  'numbs',
  'numbs',
  2,
  'CREATE TABLE numbs (Date date, No1 real, No2 real)')]
```


The Real & Float Data Types

Real data can hold a value 4 bytes in size, meaning it has 7 digits of precision (the number of digits to the right of the decimal point). It's also a floating-point numeric that is identical to the floating point statement float(24).

Like the real data type, **float data** is approximate: float can hold 8 bytes, or 15 places after the decimal point. Note that each database (MySQL, SQL Server) has different implementations. In older versions of MySQL, (pre-8.0.17) you could specify precision for float; that is, how many digits to show after the decimal point. This was done using float(*size,d*) where *size* was the total number of digits and *d* the number of digits after the decimal point.

3. See *https://www.sqlite.org/lang.html* for an overview of the SQLite3 language dialect.
Note: This is also a user friendly resource: https://www.w3schools.com/sql/
Float and real: https://docs.microsoft.com/en-us/sql/t-sql/data-types/float-and-real-transact-sql?view=sql-server-ver15

Working with SQL Databases3

- Now that there is a database file with a table, this table can be populated with data.
- Each row consists of a `datetime` object and two `float` objects

❶ Writes a single row (or record) to the `numbs` table.

❷ Creates a larger dummy data set as an `ndarray` object.

❸ Iterates over the rows of the `ndarray` object.

❹ Retrieves a number of rows from the table.

❺ The same, but with a condition on the values in the `No1` column.

❻ Defines a pointer object …

❼ … that behaves like a generator object.

❽ Retrieves all the remaining rows.

```
In [57]: import datetime

In [58]: now = datetime.datetime.now()
         q('INSERT INTO numbs VALUES(?, ?, ?)', (now, 0.12, 7.3))  ❶
Out[58]: <sqlite3.Cursor at 0x102655f80>

In [59]: np.random.seed(100)

In [60]: data = np.random.standard_normal((10000, 2)).round(4)  ❷

In [61]: %%time
         for row in data:  ❸
             now = datetime.datetime.now()
             q('INSERT INTO numbs VALUES(?, ?, ?)', (now, row[0], row[1]))
         con.commit()
         CPU times: user 115 ms, sys: 6.69 ms, total: 121 ms
         Wall time: 124 ms

In [62]: q('SELECT * FROM numbs').fetchmany(4)  ❹
Out[62]: [('2018-10-19 12:11:15.564019', 0.12, 7.3),
          ('2018-10-19 12:11:15.592956', -1.7498, 0.3427),
          ('2018-10-19 12:11:15.593033', 1.153, -0.2524),
          ('2018-10-19 12:11:15.593051', 0.9813, 0.5142)]

In [63]: q('SELECT * FROM numbs WHERE no1 > 0.5').fetchmany(4)  ❺
Out[63]: [('2018-10-19 12:11:15.593033', 1.153, -0.2524),
          ('2018-10-19 12:11:15.593051', 0.9813, 0.5142),
          ('2018-10-19 12:11:15.593104', 0.6727, -0.1044),
          ('2018-10-19 12:11:15.593134', 1.619, 1.5416)]

In [64]: pointer = q('SELECT * FROM numbs')  ❻

In [65]: for i in range(3):
             print(pointer.fetchone())  ❼
         ('2018-10-19 12:11:15.564019', 0.12, 7.3)
         ('2018-10-19 12:11:15.592956', -1.7498, 0.3427)
         ('2018-10-19 12:11:15.593033', 1.153, -0.2524)

In [66]: rows = pointer.fetchall()  ❽
         rows[:3]
Out[66]: [('2018-10-19 12:11:15.593051', 0.9813, 0.5142),
          ('2018-10-19 12:11:15.593063', 0.2212, -1.07),
          ('2018-10-19 12:11:15.593073', -0.1895, 0.255)]
```

Working with SQL Databases4

- Finally, one might want to delete the table object in the database if it’s not required anymore:

```
In [67]: q('DROP TABLE IF EXISTS numbs')  ❶
Out[67]: <sqlite3.Cursor at 0x1187a4...>

In [68]: q('SELECT * FROM sqlite_master').fetchall()  ❷
Out[68]: []

In [69]: con.close()  ❸
```

❶ Removes the table from the database.

❷ There are no table objects left after this operation.

❸ Closes the database connection.

❹ Removes the database file from disk.

SQL databases are a rather broad topic; indeed, too broad and complex to be covered in any significant way in this chapter. The basic messages are:

- Python integrates well with almost any database technology.
- The basic SQL syntax is mainly determined by the database in use; the rest is what is called “Pythonic.”

A few more examples based on SQLite3 are included later in this chapter.

Sqlite3 functions

fetchone: Fetches the next row of a query result set, returning a single sequence, or None when no more data is available.

- https://www.kite.com/python/docs/sqlite3.Cursor.fetchone

fetchall: Fetches all (remaining) rows of a query result, returning a list. Note that the cursor's arraysize attribute can affect the performance of this operation. An empty list is returned when no rows are available.

- https://www.kite.com/python/docs/sqlite3.Cursor.fetchall

fetchmany: Fetch many rows

https://www.kite.com/python/docs/sqlalchemy.engine.result.ResultProxy.fetchmany

Manual: https://docs.python.org/3/library/sqlite3.html

Writing and Reading NumPy Arrays1

- NumPy itself has functions to write and read ndarray objects in a convenient and performant fashion.
- This saves effort in some circumstances, such as when converting NumPy dtype objects into specific database data types (e.g., for SQLite3).
- To illustrate that NumPy can be an efficient replacement for a SQL-based approach, the following code replicates the example from the previous section with NumPy.
- Instead of pandas, the code uses the np.arange() function of NumPy to generate an ndarray object with datetime objects stored:


```
In [71]: dtimes = np.arange('2019-01-01 10:00:00', '2025-12-31 22:00:00',
                           dtype='datetime64[m]')  ❶

In [72]: len(dtimes)
Out[72]: 3681360

In [73]: dty = np.dtype([('Date', 'datetime64[m]'),
                         ('No1', 'f'), ('No2', 'f')])  ❷

In [74]: data = np.zeros(len(dtimes), dtype=dty)  ❸

In [75]: data['Date'] = dtimes  ❹

In [76]: a = np.random.standard_normal((len(dtimes), 2)).round(4)  ❺

In [77]: data['No1'] = a[:, 0]  ❻
         data['No2'] = a[:, 1]  ❻

In [78]: data.nbytes  ❼
Out[78]: 58901760
```

❶ Creates an ndarray object with datetime as the dtype.

❷ Defines the special dtype object for the structured array.

❸ Instantiates an ndarray object with the special dtype.

❹ Populates the Date column.

❺ The dummy data sets …

❻ … which populate the No1 and No2 columns.

❼ The size of the structured array in bytes.

Writing and Reading NumPy Arrays2

- Saving of `ndarray` objects is highly optimized and therefore quite fast.
- Almost 60 MB of data takes a fraction of a second to save on disk (here using an SSD). A larger `ndarray` object with 480 MB of data takes about half a second to save on disk:**

```
In [79]: %time np.save(path + 'array', data) ❶
         CPU times: user 37.4 ms, sys: 58.9 ms, total: 96.4 ms
         Wall time: 77.9 ms

In [80]: ll $path* ❷
         -rw-r--r--  1 yves  staff  58901888 Oct 19 12:11
          /Users/yves/Temp/data/array.npy

In [81]: %time np.load(path + 'array.npy') ❸
         CPU times: user 1.67 ms, sys: 44.8 ms, total: 46.5 ms
         Wall time: 44.6 ms

Out[81]: array([('2019-01-01T10:00',  1.5131,  0.6973),
                ('2019-01-01T10:01', -1.722 , -0.4815),
                ('2019-01-01T10:02',  0.8251,  0.3019), ...,
                ('2025-12-31T21:57',  1.372 ,  0.6446),
                ('2025-12-31T21:58', -1.2542,  0.1612),
                ('2025-12-31T21:59', -1.1997, -1.097 )],
               dtype=[('Date', '<M8[m]'), ('No1', '<f4'), ('No2', '<f4')])
```


❶ This saves the structured `ndarray` object on disk.

❷ The size on disk is hardly larger than in memory (due to binary storage).

❸ This loads the structured `ndarray` object from disk.

** Note that such times might vary significantly even on the same machine when repeated multiple times, because they depend, among other factors, on what the machine is doing CPU-wise and I/O-wise at the same time.

Writing and Reading NumPy Arrays3

```
In [82]: %time data = np.random.standard_normal((10000, 6000)).round(4)
         CPU times: user 2.69 s, sys: 391 ms, total: 3.08 s
         Wall time: 2.78 s

In [83]: data.nbytes
Out[83]: 480000000

In [84]: %time np.save(path + 'array', data)
         CPU times: user 42.9 ms, sys: 300 ms, total: 343 ms
         Wall time: 481 ms

In [85]: ll $path*
          -rw-r--r--  1 yves  staff  480000128 Oct 19 12:11
          /Users/yves/Temp/data/array.npy

In [86]: %time np.load(path + 'array.npy')

        CPU times: user 2.32 ms, sys: 363 ms, total: 365 ms
        Wall time: 363 ms

Out[86]: array([[ 0.3066,  0.5951,  0.5826, ...,  1.6773,  0.4294, -0.2216],
                [ 0.8769,  0.7292, -0.9557, ...,  0.5084,  0.9635, -0.4443],
                [-1.2202, -2.5509, -0.0575, ..., -1.6128,  0.4662, -1.3645],
                ...,
                [-0.5598,  0.2393, -2.3716, ...,  1.7669,  0.2462,  1.035 ],
                [ 0.273 ,  0.8216, -0.0749, ..., -0.0552, -0.8396,  0.3077],
                [-0.6305,  0.8331,  1.3702, ...,  0.3493,  0.1981,  0.2037]])
```

❹ A larger regular `ndarray` object.

Writing and Reading NumPy Arrays

- These examples illustrate that writing to disk in this case is mainly hardware-bound, since the speeds observed represent roughly the advertised writing speed of standard SSDs at the time of this writing (about 500 MB/s).
- In any case, one can expect that this form of data storage and retrieval is faster when compared to SQL databases or using the `pickle` module for serialization.
- There are two reasons: first, the data is mainly numeric; second, `NumPy` uses binary storage, which reduces the overhead almost to zero.
- Of course, one does not have the functionality of a SQL database available with this approach, but `PyTables` will help in this regard, as subsequent sections show.
- Note there is an extra section showing you how to use function np.savez_compressed() which creates a compressed numpy array. Writing and reading takes slightly longer.

I/O with Pandas1

One of the major strengths of `pandas` is that it can read and write different data formats natively, including:

- CSV (comma-separated values)
- SQL (Structured Query Language)
- XLS/XSLX (Microsoft Excel files)
- JSON (JavaScript Object Notation)
- HTML (HyperText Markup Language)

Table 9-1 lists the supported formats and the corresponding import and export functions/methods of `pandas` and the `DataFrame` class, respectively.

The parameters that, for example, the `pd.read_csv()` import function takes are described in the documentation for `pandas.read_csv`.

Table 9-1. Import-export functions and methods

Format	Input	Output	Remark
CSV	pd.read_csv()	.to_csv()	Text file
XLS/XLSX	pd.read_excel()	.to_excel()	Spreadsheet
HDF	pd.read_hdf()	.to_hdf()	HDF5 database
SQL	pd.read_sql()	.to_sql()	SQL table
JSON	pd.read_json()	.to_json()	JavaScript Object Notation
MSGPACK	pd.read_msgpack()	.to_msgpack()	Portable binary format
HTML	pd.read_html()	.to_html()	HTML code
GBQ	pd.read_gbq()	.to_gbq()	Google Big Query format
DTA	pd.read_stata()	.to_stata()	Formats 104, 105, 108, 113-115, 117
Any	pd.read_clipboard()	.to_clipboard()	E.g., from HTML page
Any	pd.read_pickle()	.to_pickle()	(Structured) Python object

I/O with Pandas2

- The test case is again a larger set of `float` objects:

```
In [88]: data = np.random.standard_normal((1000000, 5)).round(4)

In [89]: data[:3]
Out[89]: array([[ 0.4918,  1.3707,  0.137 ,  0.3981, -1.0059],
                [ 0.4516,  1.4445,  0.0555, [illegible],  0.44  ],
                [ 0.1629, -0.8473, -0.8223, -0.4621, -0.5137]])
```

- To this end, this section also revisits SQLite3 and compares the performance to alternative formats using `pandas`.

Working with SQL Databases1

- All that follows with regard to SQLite3 should be familiar by now:

```
In [90]: filename = path + 'numbers'

In [91]: con = sq3.Connection(filename + '.db')

In [92]: query = 'CREATE TABLE numbers (No1 real, No2 real,\
                 No3 real, No4 real, No5 real)'  ❶

In [93]: q = con.execute
         qm = con.executemany

In [94]: q(query)
Out[94]: <sqlite3.Cursor at 0x1187a76c0>
```


❶ Creates a table with five columns for real numbers (`float` objects).

Working with SQL Databases2

- This time, the `.executemany()` method can be applied since the data is available in a single `ndarray` object.
- Reading and working with the data works as before.
- Query results can also be visualized easily (see Figure 9-1):

```
In [95]: %%time
         qm('INSERT INTO numbers VALUES (?, ?, ?, ?, ?)', data)  ❶
         con.commit()
         CPU times: user 7.3 s, sys: 195 ms, total: 7.49 s
         Wall time: 7.71 s

In [96]: ll $path*
         -rw-r--r--  1 yves  staff  52633600 Oct 19 12:11
          /Users/yves/Temp/data/numbers.db

In [97]: %%time
         temp = q('SELECT * FROM numbers').fetchall()  ❷
         print(temp[:3])
         [(0.4918, 1.3707, 0.137, 0.3981, -1.0059), (0.4516, 1.4445, 0.0555
          -0.0397, 0.44), (0.1629, -0.8473, -0.8223, -0.4621, -0.5137)]
         CPU times: user 1.7 s, sys: 124 ms, total: 1.82 s
         Wall time: 1.9 s

   [98]: %%time
         query = 'SELECT * FROM numbers WHERE No1 > 0 AND No2 < 0'
         res = np.array(q(query).fetchall()).round(3)  ❸
         CPU times: user 639 ms, sys: 64.7 ms, total: 704 ms
         Wall time: 702 ms

   [99]: res = res[::100]  ❹
         plt.figure(figsize=(10, 6))
         plt.plot(res[:, 0], res[:, 1], 'ro')  ❹
```


❶ Inserts the whole data set into the table in a single step.

❷ Retrieves all the rows from the table in a single step.

❸ Retrieves a selection of the rows and transforms it to an `ndarray` object.

❹ Plots a subset of the query result.

Working with SQL Databases3

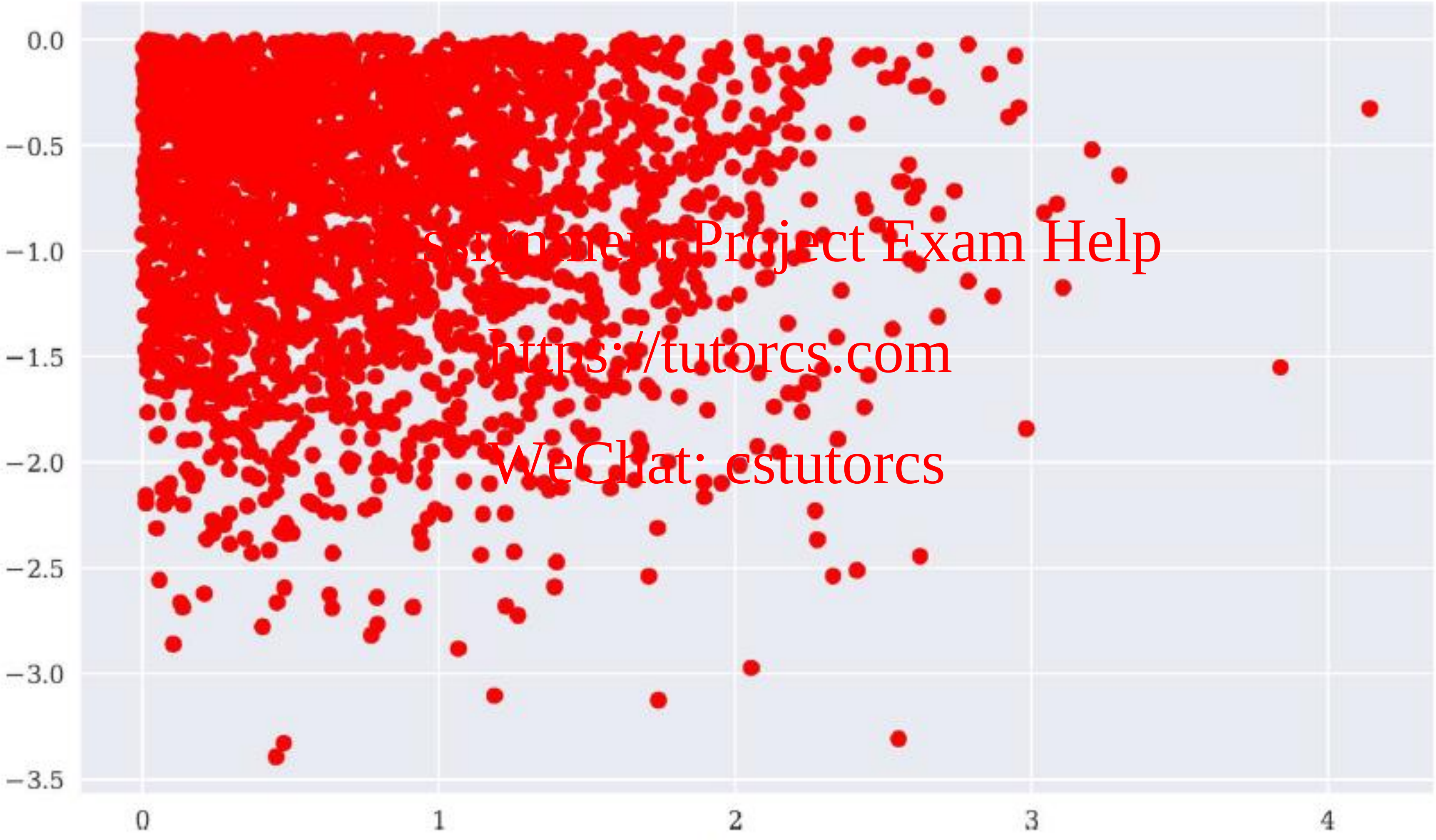


Figure 9-1. Scatter plot of the query result (selection)

From SQL to pandas1

- A generally more efficient approach, however, is the reading of either whole tables or query results with `pandas`.
- When one can read a whole table into memory, analytical queries can generally be executed much faster than when using the SQL disk-based approach (out-of-memory).
- Reading the whole table with `pandas` takes roughly the same amount of time as reading it into a `NumPy` `ndarray` object.
- There as here, the bottleneck performance-wise is the SQL database:


```
In [100]: %time data = pd.read_sql('SELECT * FROM numbers', con)  ❶
          CPU times: user 2.17 s, sys: 180 ms, total: 2.35 s
          Wall time: 2.32 s

In [101]: data.head()
Out[101]:       No1     No2     No3     No4     No5
          0  0.4918  1.3707  0.1370  0.3981 -1.0059
          1  0.4516  1.4445  0.0555 -0.0397  0.4400
          2  0.1629 -0.8473 -0.8223 -0.4621 -0.5137
          3  1.3064  0.9125  0.5142 -0.7868 -0.3398
          4 -0.1148 -1.5215 -0.7045 -1.0042 -0.0600
```

❶ Reads all rows of the table into the `DataFrame` object named `data`.

From SQL to pandas2

- The data is now in-memory, which allows for much faster analytics.
- The speedup is often an order of magnitude or more. `pandas` can also master more complex queries, although it is neither meant nor able to replace SQL databases when it comes to complex relational data structures.
- The result of the query with multiple conditions combined is shown in Figure 9-2:

```
In [102]: %time data[(data['No1'] > 0) & (data['No2'] < 0)].head()  ❶
          CPU times: user 47.1 ms, sys: 12.3 ms, total: 59.4 ms
          Wall time: 33.4 ms

Out[102]:          No1      No2      No3      No4      No5
          2    0.1629  -0.8473  -0.8223  -0.4621  -0.5137
          5    0.1893  -0.0207  -0.2104   0.9419   0.2551
          8    1.4784  -0.3333  -0.7050   0.3586  -0.3937
          10   0.8092  -0.9899   1.0364  -1.0453   0.0579
          11   0.9065  -0.7757  -0.9267   0.7797   0.0863

In [103]: %%time
          q = '(No1 < -0.5 | No1 > 0.5) & (No2 < -1 | No2 > 1)'  ❷
          res = data[['No1', 'No2']].query(q)  ❷
          CPU times: user 95.4 ms, sys: 22.4 ms, total: 118 ms
          Wall time: 56.4 ms

In [104]: plt.figure(figsize=(10, 6))
          plt.plot(res['No1'], res['No2'], 'ro');
```


❶ Two conditions combined logically.

❷ Four conditions combined logically.

From SQL to pandas3

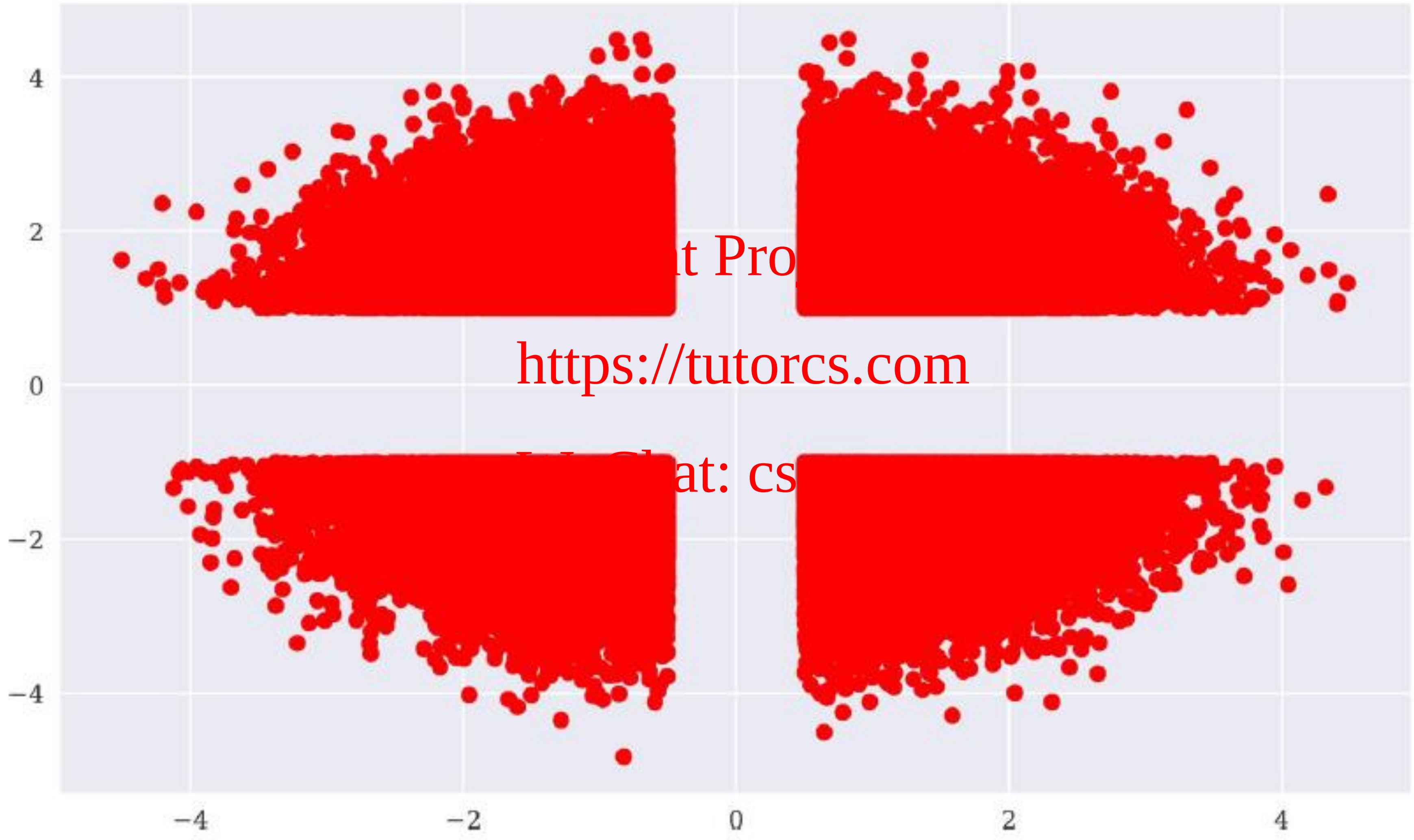


Figure 9-2. Scatter plot of the query result (selection)

From SQL to pandas4

- As expected, using the in-memory analytics capabilities of `pandas` leads to a significant speedup, provided `pandas` is able to replicate the respective SQL statement.
- This is not the only advantage of using `pandas`, since `pandas` is tightly integrated with a number of other packages (including `PyTables`, the topic of the subsequent section).
- Here, it suffices to know that the combination of both can speed up I/O operations considerably.
- This is shown in the following code:

```
In [105]: h5s = pd.HDFStore(filename + '.h5s', 'w')  ❶

In [106]: %time h5s['data'] = data  ❷
          CPU times: user 46.7 ms, sys: 47.1 ms, total: 93.8 ms
          Wall time: 99.7 ms

In [107]: h5s  ❸
Out[107]: <class 'pandas.io.pytables.HDFStore'>
          File path: /Users/yves/Temp/data/numbers.h5s

In [108]: h5s.close()  ❹
```


❶ This opens an HDF5 database file for writing; in `pandas` an `HDFStore` object is created.

❷ The complete `DataFrame` object is stored in the database file via binary storage.

❸ The `HDFStore` object information.

❹ The database file is closed.

From SQL to pandas5

- The whole `DataFrame` with all the data from the original SQL table is written much faster when compared to the same procedure with SQLite3.
- Reading is even faster:

```
In [109]: %%time
          h5s = pd.HDFStore(filename + '.h5s', 'r')  ❶
          data_ = h5s['data']  ❷
          h5s.close()  ❸
          CPU times: user 11 ms, sys: 18.3 ms, total: 29.3 ms
          Wall time: 29.4 ms

In [110]: data_ is data  ❹
Out[110]: False

In [111]: (data_ == data).all()  ❺
Out[111]: No1    True
          No2    True
          No3    True
          No4    True
          No5    True
          dtype: bool

In [112]: np.allclose(data_, data)  ❺
Out[112]: True
```

❶ This opens the HDF5 database file for reading.

❷ The `DataFrame` is read and stored in-memory as `data_`.

❸ The database file is closed.

❹ The two `DataFrame` objects are not the same …

❺ … but they now contain the same data.

Working with Csv Files1

- One of the most widely used formats to exchange financial data is the CSV format.
- Although it is not really standardized, it can be processed by any platform and the vast majority of applications concerned with data and financial analytics.
- Earlier, we saw how to write and read data to and from CSV files with standard Python functionality (see "Reading and Writing Text Files").
- `pandas` makes this whole procedure a bit more convenient, the code more concise, and the execution in general faster (see also Figure 9-3):

```
In [114]: %time data.to_csv(filename + '.csv')  ❶
          CPU times: user 6.44 s, sys: 139 ms, total: 6.58 s
          Wall time: 6.71 s

In [115]: ll $path
          total 283672
          -rw-r--r--  1 yves  staff  43834157 Oct 19 12:11 numbers.csv
          -rw-r--r--  1 yves  staff  52633600 Oct 19 12:11 numbers.db
          -rw-r--r--  1 yves  staff  48007240 Oct 19 12:11 numbers.h5s

In [116]: %time df = pd.read_csv(filename + '.csv')  ❷
          CPU times: user 1.12 s, sys: 111 ms, total: 1.23 s
          Wall time: 1.23 s

In [117]: df[['No1', 'No2', 'No3', 'No4']].hist(bins=20, figsize=(10, 6));
```


❶ The `.to_csv()` method writes the `DataFrame` data to disk in CSV format.

❷ The `pd.read_csv()` method then reads it back into memory as a new `DataFrame` object.

Working with Csv Files2

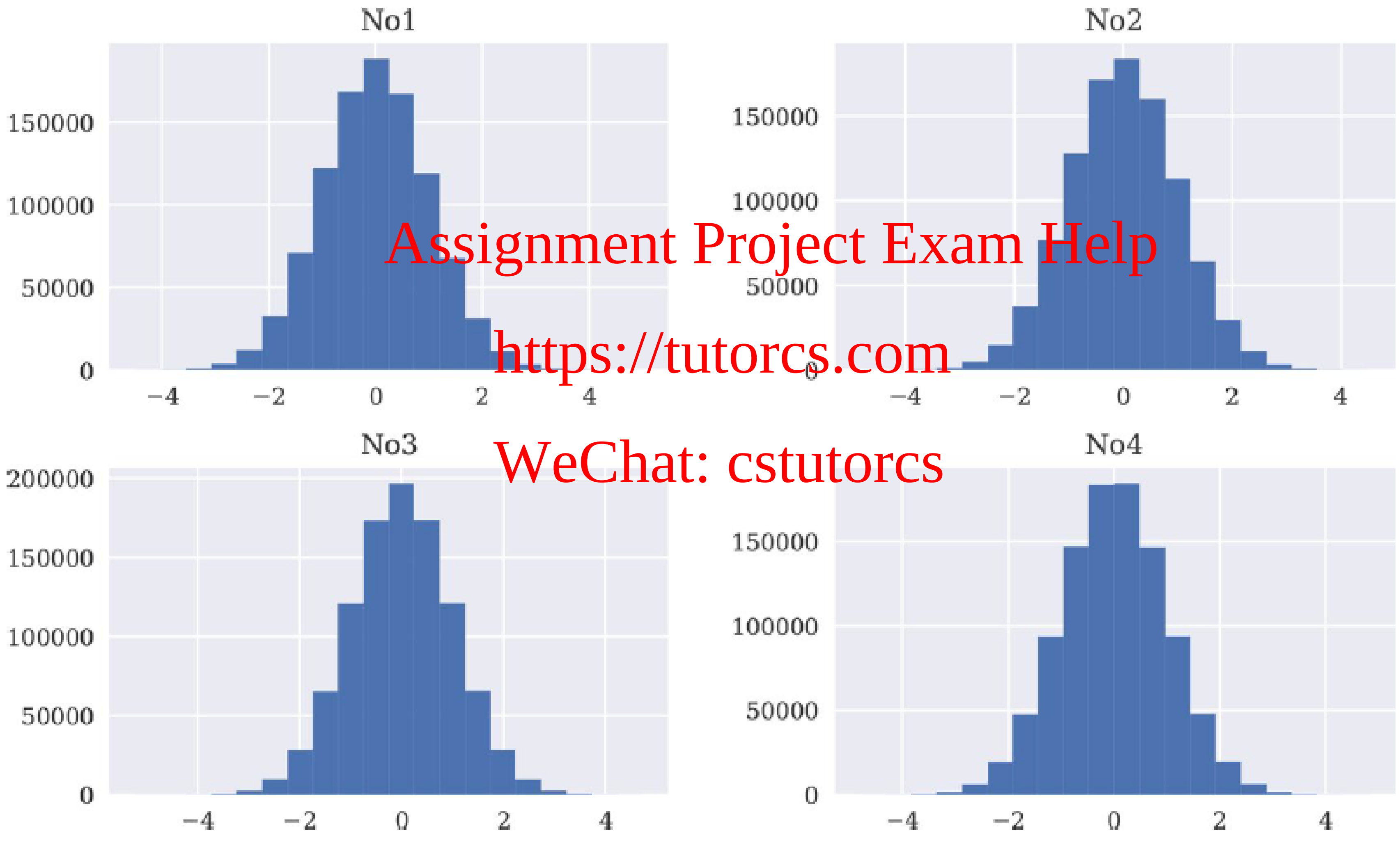


Figure 9-3. Histograms for selected columns

Working with Excel Files1

- The following code briefly demonstrates how `pandas` can write data in Excel format and read data from Excel spreadsheets. In this case, the data set is restricted to 100,000 rows (see also Figure 9-4):

```
In [118]: %time data[:100000].to_excel(filename + '.xlsx')  ❶
          CPU times: user 25.9 s, sys: 520 ms, total: 26.4 s
          Wall time: 27.3 s

In [119]: %time df = pd.read_excel(filename + '.xlsx', 'Sheet1')  ❷
          CPU times: user 5.78 s, sys: 70.1 ms, total: 5.85 s
          Wall time: 5.91 s

In [120]: df.cumsum().plot(figsize=(10, 6));
```


❶

The `.to_excel()` method writes the `DataFrame` data to disk in XLSX format.

❷

The `pd.read_excel()` method then reads it back into memory as a new `DataFrame` object, also specifying the sheet from which to read.

Working with Excel Files2

- Generating the Excel spreadsheet file with a smaller subset of the data takes quite a while.
- This illustrates what kind of overhead the spreadsheet structure brings along with it.
- Inspection of the generated files reveals that the `DataFrame` with `HDFStore` combination is the most compact alternative (using compression, as described in the next section, further increases the benefits). The same amount of data as a CSV file — i.e., as a text file — is somewhat larger in size.
- This is one reason for the slower performance when working with CSV files, the other being the very fact that they are “only” general text files.

Working with Excel Files3

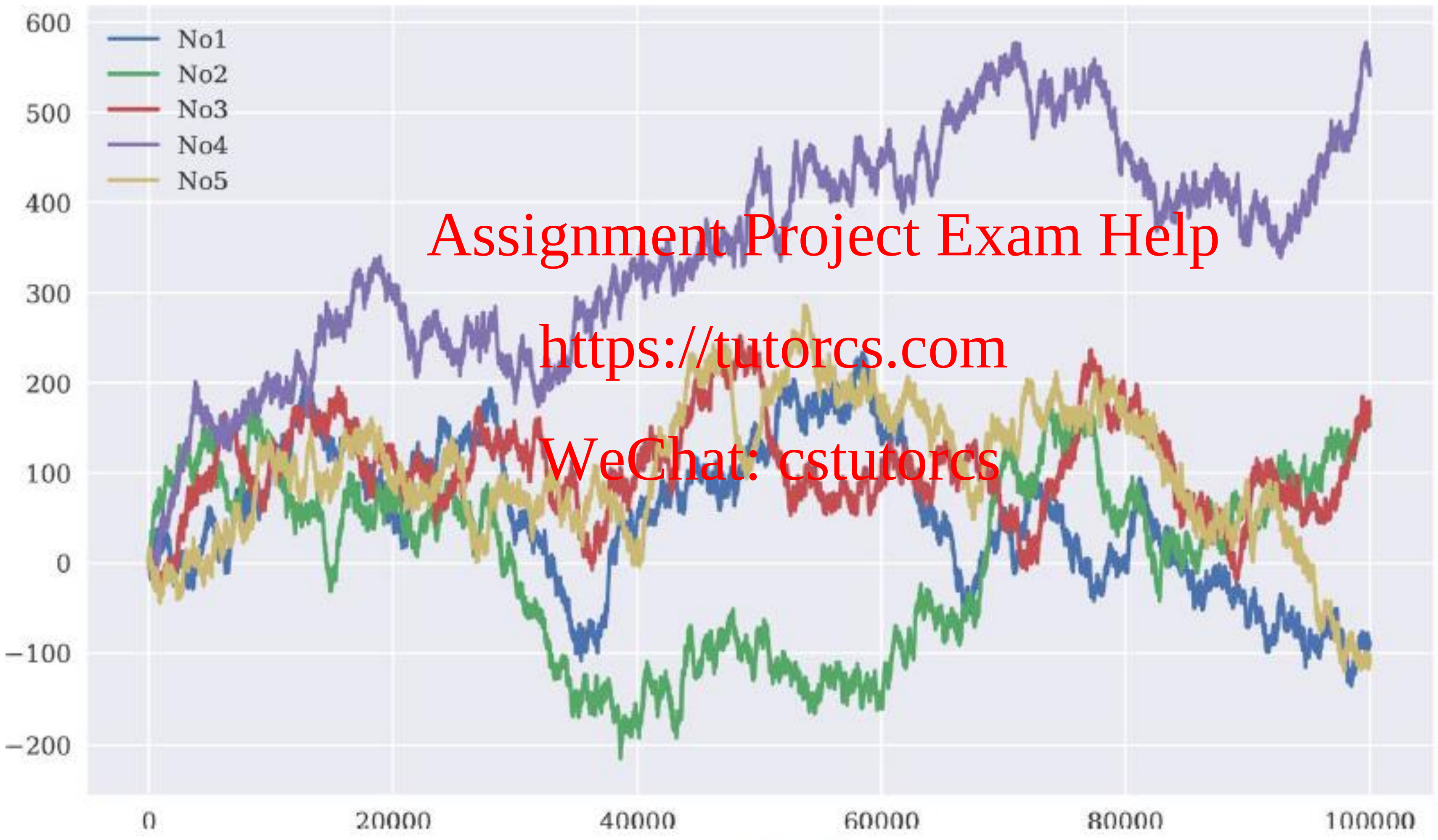


Figure 9-4. Line plots for all columns

I/O with PyTables

- `PyTables` is a Python binding for the HDF5 database standard.
- It is specifically designed to optimize the performance of I/O operations and make best use of the available hardware. The library's import name is `tables`.
- Similar to `pandas`, when it comes to in-memory analytics `PyTables` is neither able nor meant to be a full replacement for SQL databases.
- However, it brings along some features that further close the gap.
- For example, a `PyTables` database can have many tables, and it supports compression and indexing and also nontrivial queries on tables.
- In addition, it can store `NumPy` arrays efficiently and has its own flavor of array-like data structures.
- To begin with, some imports:


```
In [123]: import tables as tb  ❶
          import datetime as dt
```

❶ The package name is `PyTables`, the import name is `tables`.

Working with Tables1

- `PyTables` provides a file-based database format, similar to SQLite3.**
- The following opens a database file and creates a table: (see next slide)

** Many other databases require a server/client architecture.
For interactive data and financial analytics, file-based databases prove a bit more convenient and also sufficient for most purposes.

Working with Tables2

❶ Opens the database file in HDF5 binary storage format.

❷ The `Date` column for date-time information (as a `str` object).

❸ The two columns to store `int` objects.

❹ The two columns to store `float` objects.

❺ Via `Filters` objects, compression levels can be specified, among other things.

❻ The node (path) and technical name of the table.

❼ The description of the row data structure.

❽ The name (title) of the table.

❾ The expected number of rows; allows for optimizations.

❿ The `Filters` object to be used for the table.

```
In [124]: filename = path + 'pytab.h5'

In [125]: h5 = tb.open_file(filename, 'w')  ❶

In [126]: row_des = {
              'Date': tb.StringCol(26, pos=1),  ❷
              'No1': tb.IntCol(pos=2),  ❸
              'No2': tb.IntCol(pos=3),  ❸
              'No3': tb.Float64Col(pos=4),  ❹
              'No4': tb.Float64Col(pos=5)  ❹
              }

In [127]: rows = 2000000

In [128]: filters = tb.Filters(complevel=0)  ❺

In [129]: tab = h5.create_table('/', 'ints_floats',  ❻
                                row_des,  ❼
                                title='Integers and Floats',  ❽
                                expectedrows=rows,  ❾
                                filters=filters)  ❿

In [130]: type(tab)
Out[130]: tables.table.Table

In [131]: tab
Out[131]: /ints_floats (Table(0,)) 'Integers and Floats'
            description := {
            "Date": StringCol(itemsize=26, shape=(), dflt=b'', pos=0),
            "No1": Int32Col(shape=(), dflt=0, pos=1),
            "No2": Int32Col(shape=(), dflt=0, pos=2),
            "No3": Float64Col(shape=(), dflt=0.0, pos=3),
            "No4": Float64Col(shape=(), dflt=0.0, pos=4)}
            byteorder := 'little'
            chunkshape := (2621,)
```


https://www.pytables.org/usersguide/libref/helper_classes.html?highlight=complevel#tables.Filters.complevel

Working with Tables3

- To populate the table with numerical data, two `ndarray` objects with random numbers are generated: one with random integers, the other with random floating-point numbers.
- The population of the table happens via a simple Python loop:

```
In [132]: pointer = tab.row  ❶

In [133]: ran_int = np.random.randint(0, 10000, size=(rows, 2))  ❷

In [134]: ran_flo = np.random.standard_normal((rows, 2)).round(4)  ❸

In [135]: %%time
          for i in range(rows):
              pointer['Date'] = dt.datetime.now()  ❹
              pointer['No1'] = ran_int[i, 0]  ❹
              pointer['No2'] = ran_int[i, 1]  ❹
              pointer['No3'] = ran_flo[i, 0]  ❹
              pointer['No4'] = ran_flo[i, 1]  ❹
              pointer.append()  ❺
          tab.flush()  ❻
          CPU times: user 8.16 s, sys: 78.7 ms, total: 8.24 s
          Wall time: 8.25 s

In [136]: tab  ❼
Out[136]: /ints_floats (Table(2000000,)) 'Integers and Floats'
            description := {
            "Date": StringCol(itemsize=26, shape=(), dflt=b'', pos=0),
            "No1": Int32Col(shape=(), dflt=0, pos=1),
            "No2": Int32Col(shape=(), dflt=0, pos=2),
            "No3": Float64Col(shape=(), dflt=0.0, pos=3),
            "No4": Float64Col(shape=(), dflt=0.0, pos=4)}
            byteorder := 'little'
            chunkshape := (2621,)
```


❶ A pointer object is created.

❷ The `ndarray` object with the random `int` objects is created.

❸ The `ndarray` object with the random `float` objects is created.

❹ The `datetime` object and the two `int` and two `float` objects are written row-by-row.

❺ The new row is appended.

❻ All written rows are flushed; i.e., committed as permanent changes.

❼ The changes are reflected in the `Table` object description.

Working with Tables4

- The Python loop is quite slow in this case.
- There is a more performant and Pythonic way to accomplish the same result, by the use of NumPy structured arrays.
- Equipped with the complete data set stored in a structured array, the creation of the table boils down to a single line of code.
- Note that the row description is not needed anymore; PyTables uses the dtype object of the structured array to infer the data types instead:

❶ This defines the special dtype object.

❷ This creates the structured array with zeros (and empty strings).

❸ A few records from the ndarray object.

❹ The columns of the ndarray object are populated at once.

❺ This creates the Table object *and* populates it with the data.

```
In [138]: dty = np.dtype([('Date', 'S26'), ('No1', '<i4'), ('No2', '<i4'),
                          ('No3', '<f8'), ('No4', '<f8')])  ❶

In [139]: sarray = np.zeros(len(ran_int), dtype=dty)  ❷

In [140]: sarray[:4]  ❸
Out[140]: array([(b'', 0, 0, 0., 0.), (b'', 0, 0, 0., 0.), (b'', 0, 0, 0., 0.),
                 (b'', 0, 0, 0., 0.)],
          dtype=[('Date', 'S26'), ('No1', '<i4'), ('No2', '<i4'), ('No3', '<f8'),
           ('No4', '<f8')])

In [141]: %%time
          sarray['Date'] = dt.datetime.now()  ❹
          sarray['No1'] = ran_int[:, 0]  ❹
          sarray['No2'] = ran_int[:, 1]  ❹
          sarray['No3'] = ran_flo[:, 0]  ❹
          sarray['No4'] = ran_flo[:, 1]  ❹
          CPU times: user 161 ms, sys: 42.7 ms, total: 204 ms
          Wall time: 207 ms

In [142]: %%time
          h5.create_table('/', 'ints_floats_from_array', sarray,
                                title='Integers and Floats',
                                expectedrows=rows, filters=filters)  ❺
          CPU times: user 42.9 ms, sys: 51.4 ms, total: 94.3 ms
          Wall time: 96.6 ms

Out[142]: /ints_floats_from_array (Table(2000000,)) 'Integers and Floats'
            description := {
            "Date": StringCol(itemsize=26, shape=(), dflt=b'', pos=0),
            "No1": Int32Col(shape=(), dflt=0, pos=1),
            "No2": Int32Col(shape=(), dflt=0, pos=2),
            "No3": Float64Col(shape=(), dflt=0.0, pos=3),
            "No4": Float64Col(shape=(), dflt=0.0, pos=4)}
            byteorder := 'little'
            chunkshape := (2621,)
```

Working with Tables5

```
In [143]: type(h5)
Out[143]: tables.file.File
```

```
In [144]: h5  ❶
Out[144]: File(filename=/Users/yves/Temp/data/pytab.h5, title='', mode='w',
           root_uep='/', filters=Filters(complevel=0, shuffle=False,
           bitshuffle=False, fletcher32=False, least_significant_digit=None))
          / (RootGroup) ''
          /ints_floats (Table(2000000,)) 'Integers and Floats'
            description := {
            "Date": StringCol(itemsize=26, shape=(), dflt=b'', pos=0),
            "No1": Int32Col(shape=(), dflt=0, pos=1),
            "No2": Int32Col(shape=(), dflt=0, pos=2),
            "No3": Float64Col(shape=(), dflt=0.0, pos=3),
            "No4": Float64Col(shape=(), dflt=0.0, pos=4)}
            byteorder := 'little'
            chunkshape := (2621,)
          /ints_floats_from_array (Table(2000000,)) 'Integers and Floats'
            description := {
            "Date": StringCol(itemsize=26, shape=(), dflt=b'', pos=0),
            "No1": Int32Col(shape=(), dflt=0, pos=1),
            "No2": Int32Col(shape=(), dflt=0, pos=2),
            "No3": Float64Col(shape=(), dflt=0.0, pos=3),
            "No4": Float64Col(shape=(), dflt=0.0, pos=4)}
            byteorder := 'little'
            chunkshape := (2621,)
```

```
In [145]: h5.remove_node('/', 'ints_floats_from_array')  ❷
```

- This approach is an order of magnitude faster, has more concise code, and achieves the same result:

❶ The description of the `File` object with the two `Table` objects.

❷ This removes the second `Table` object with the redundant data.

Working with Tables6

- The `Table` object behaves pretty similar to `NumPy` structured `ndarray` objects in most cases (see also Figure 9-5):

```
In [146]: tab[:3] ❶
Out[146]: array([(b'2018-10-19 12:12:28.227771', 8576, 5991, -0.0528, 0.2468),
                 (b'2018-10-19 12:12:28.227858', 2990, 9310, -0.0261, 0.3932),
                 (b'2018-10-19 12:12:28.227868', 4400, 4823,  0.9133, 0.2579)],
          dtype=[('Date', 'S26'), ('No1', '<i4'), ('No2', '<i4'), ('No3', '<f8'),
           ('No4', '<f8')])

In [147]: tab[:4]['No4'] ❷
Out[147]: array([ 0.2468,  0.3932,  0.2579, -0.5582])

In [148]: %time np.sum(tab[:]['No3']) ❸
          CPU times: user 76.7 ms, sys: 74.8 ms, total: 151 ms
          Wall time: 152 ms

Out[148]: 88.8542999999997

In [149]: %time np.sum(np.sqrt(tab[:]['No1'])) ❸
          CPU times: user 91 ms, sys: 57.9 ms, total: 149 ms
          Wall time: 104 ms

Out[149]: 133349920.3689251

In [150]: %%time
          plt.figure(figsize=(10, 6))
          plt.hist(tab[:]['No3'], bins=30); ❹
          CPU times: user 328 ms, sys: 72.1 ms, total: 400 ms
          Wall time: 456 ms
```


❶ Selecting rows via indexing.

❷ Selecting column values only via indexing.

❸ Applying `NumPy` universal functions.

❹ Plotting a column from the `Table` object.

Working with Tables7

Figure 9-5. Histogram of column data

Working with Tables8

- `PyTables` also provides flexible tools to query data via typical SQL-like statements, as in the following example (the result of which is illustrated in Figure 9-6; compare it with Figure 9-2, based on a `pandas` query):

```
In [151]: query = '((No3 < -0.5) | (No3 > 0.5)) & ((No4 < -1) | (No4 > 1))'  ❶

In [152]: iterator = tab.where(query)  ❷

In [153]: %time res = [(row['No3'], row['No4']) for row in iterator]  ❸
          CPU times: user 269 ms, sys: 64.4 ms, total: 333 ms
          Wall time: 294 ms

In [154]: res = np.array(res)  ❹
          res[:3]
Out[154]: array([[0.7694, 1.4866],
                 [0.9201, 1.3346],
                 [1.4701, 1.8776]])

55]: plt.figure(figsize=(10, 6))
     plt.plot(res.T[0], res.T[1], 'ro');
```


❶ The query as a `str` object, four conditions combined by logical operators.

❷ The iterator object based on the query.

❸ The rows resulting from the query are collected via a list comprehension …

❹ … and transformed to an `ndarray` object.

Working with Tables9

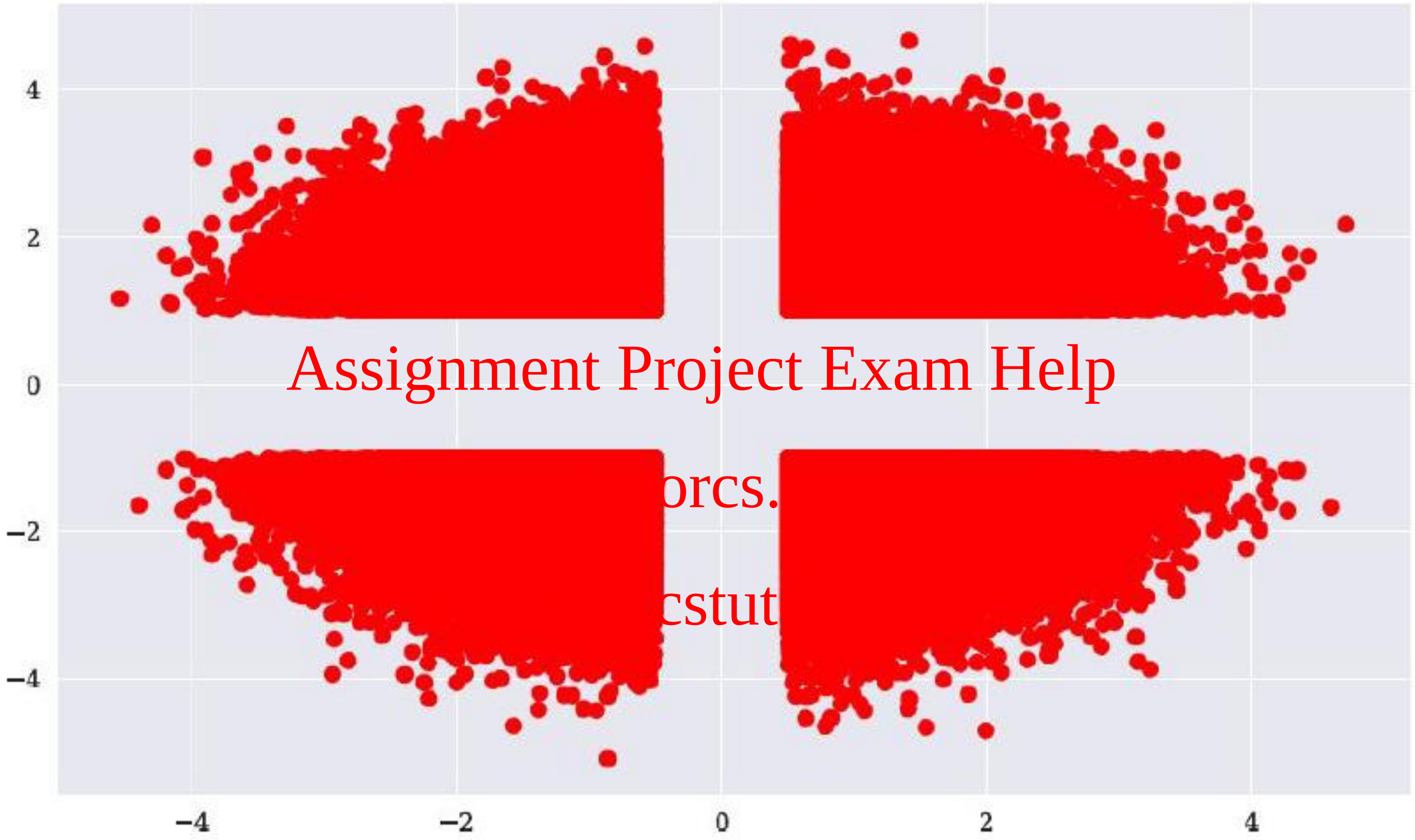


Figure 9-6. Histogram of column data

Working with Tables10

- As the following examples show, working with data stored in `PyTables` as `Table` objects gives the impression of working with `NumPy` or `pandas` objects in-memory, both from a *syntax* and a *performance* point of view:

Assignment Project Exam Help

https://tutorcs.com

WeChat: cstutorcs

FAST QUERIES

Both `pandas` and `PyTables` are able to process relatively complex, SQL-like queries and selections. They are both optimized for speed when it comes to such operations. Although there are limits to these approaches compared to relational databases, for most numerical and financial applications these are often not relevant.

```
In [156]: %%time
          values = tab[:]['No3']
          print('Max %18.3f' % values.max())
          print('Ave %18.3f' % values.mean())
          print('Min %18.3f' % values.min())
          print('Std %18.3f' % values.std())
          Max              5.224
          Ave              0.000
          Min             -5.649
          Std              1.000
          CPU times: user 163 ms, sys: 70.4 ms, total: 233 ms
          Wall time: 234 ms

In [157]: %%time
          res = [(row['No1'], row['No2']) for row in
                 tab.where('((No1 > 9800) | (No1 < 200)) \
                          & ((No2 > 4500) & (No2 < 5500))')]
          CPU times: user 165 ms, sys: 52.5 ms, total: 218 ms
          Wall time: 155 ms

In [158]: for r in res[:4]:
              print(r)
          (91, 4870)
          (9803, 5026)
          (9846, 4859)
          (9823, 5069)

In [159]: %%time
          res = [(row['No1'], row['No2']) for row in
                  tab.where('(No1 == 1234) & (No2 > 9776)')]
          CPU times: user 58.9 ms, sys: 40.5 ms, total: 99.4 ms
          Wall time: 81 ms

In [160]: for r in res:
              print(r)
          (1234, 9841)
          (1234, 9821)
          (1234, 9867)
          (1234, 9987)
          (1234, 9849)
          (1234, 9800)
```

Working with Compressed Tables1

- A major advantage of working with `PyTables` is the approach it takes to compression.
- It uses compression not only to save space on disk, but also to improve the performance of I/O operations in certain hardware scenarios.
- How does this work? When I/O is the bottleneck and the CPU is able to (de)compress data fast, the net effect of compression in terms of speed might be positive.
- Since the following examples are based on the I/O of a standard SSD, there is no speed advantage of compression to be observed.
- However, there is also almost no *disadvantage* to using compression:
- Blose Compression Engine info: https://www.blosc.org/
- https://www.pytables.org/usersguide/libref/helper_classes.html?highlight=complib#tables.Filters.complib

```
In [161]: filename = path + 'pytabc.h5'

In [162]: h5c = tb.open_file(filename, 'w')

In [163]: filters = tb.Filters(complevel=5,       ❶
                               complib='blosc')   ❷

In [164]: tabc = h5c.create_table('/', 'ints_floats', sarray,
                                  title='Integers and Floats',
                                  expectedrows=rows, filters=filters)

In [165]: query = '((No3 < -0.5) | (No3 > 0.5)) & ((No4 < -1) | (No4 > 1))'

In [166]: iteratorc = tabc.where(query)   ❸

In [167]: %time res = [(row['No3'], row['No4']) for row in iteratorc]   ❹
          CPU times: user 300 ms, sys: 50.8 ms, total: 351 ms
          Wall time: 311 ms

In [168]: res = np.array(res)
          res[:3]

Out[168]: array([[0.7694, 1.4866],
                 [0.9201, 1.3346],
                 [1.4701, 1.8776]])
```

❶

The `complevel` (compression level) parameter can take values between `0` (no compression) and `9` (highest compression).

❷

The Blosc compression engine is used, which is optimized for performance.

❸

This creates the iterator object, based on the query from before.

❹

The rows resulting from the query are collected via a list comprehension.

Working with Compressed Tables2

- Generating the compressed `Table` object with the original data and doing analytics on it is slightly slower compared to the uncompressed `Table` object.
- What about reading the data into an `ndarray` object?
- Let's check:

```
In [169]: %time arr_non = tab.read()  ❶
          CPU times: user 63 ms, sys: 78.5 ms, total: 142 ms
          Wall time: 149 ms

In [170]: tab.size_on_disk
Out[170]: 100122200

In [171]: arr_non.nbytes
Out[171]: 100000000

In [172]: %time arr_com = tabc.read()  ❷
          CPU times: user 106 ms, sys: 55.5 ms, total: 161 ms
          Wall time: 173 ms

In [173]: tabc.size_on_disk
Out[173]: 41306140

In [174]: arr_com.nbytes
Out[174]: 100000000

In [175]: ll $path*  ❸
          -rw-r--r--  1 yves  staff  200312336 Oct 19 12:12
           /Users/yves/Temp/data/pytab.h5
          -rw-r--r--  1 yves  staff   41341436 Oct 19 12:12
           /Users/yves/Temp/data/pytabc.h5

In [176]: h5c.close()  ❹
```


❶ Reading from the uncompressed `Table` object `tab`.

❷ Reading from the compressed `Table` object `tabc`.

❸ Comparing the sizes — the size of the compressed table is significantly reduced.

❹ Closing the database file.

Working with Compressed Tables3

The examples show that there is hardly any speed difference when working with compressed `Table` objects as compared to uncompressed ones.

However, file sizes on disk might — depending on the quality of the data — be significantly reduced, which has a number of benefits:

- Storage costs are reduced.
- Backup costs are reduced.
- Network traffic is reduced.
- Network speed is improved (storage on and retrieval from remote
- servers is faster).
- CPU utilization is increased to overcome I/O bottlenecks.

Working with Arrays1

- "Basic I/O with Python" showed that NumPy has built-in fast writing and reading capabilities for ndarray objects.
- PyTables is also quite fast and efficient when it comes to storing and retrieving ndarray objects, and since it is based on a hierarchical database structure, many convenience features come on top:

```
In [177]: %%time
          arr_int = h5.create_array('/', 'integers', ran_int)  ❶
          arr_flo = h5.create_array('/', 'floats', ran_flo)  ❷
          CPU times: user 4.26 ms, sys: 37.2 ms, total: 41.5 ms
          Wall time: 46.2 ms

In [178]: h5  ❸
Out[178]: File(filename=/Users/yves/Temp/data/pytab.h5, title='', mode='w',
           root_uep='/', filters=Filters(complevel=0, shuffle=False,
           bitshuffle=False, fletcher32=False, least_significant_digit=None))
          / (RootGroup) ''
          /floats (Array(2000000, 2)) ''
            atom := Float64Atom(shape=(), dflt=0.0)
            maindim := 0
            flavor := 'numpy'
            byteorder := 'little'
            chunkshape := None
          /integers (Array(2000000, 2)) ''
            atom := Int64Atom(shape=(), dflt=0)
            maindim := 0
            flavor := 'numpy'
            byteorder := 'little'
            chunkshape := None
          /ints_floats (Table(2000000,)) 'Integers and Floats'
            description := {
            "Date": StringCol(itemsize=26, shape=(), dflt=b'', pos=0),
            "No1": Int32Col(shape=(), dflt=0, pos=1),
            "No2": Int32Col(shape=(), dflt=0, pos=2),
            "No3": Float64Col(shape=(), dflt=0.0, pos=3),
            "No4": Float64Col(shape=(), dflt=0.0, pos=4)}
            byteorder := 'little'
            chunkshape := (2621,)

In [179]: ll $path*
          -rw-r--r--  1 yves  staff  262344490 Oct 19 12:12
           /Users/yves/Temp/data/pytab.h5
          -rw-r--r--  1 yves  staff  41341436 Oct 19 12:12
           /Users/yves/Temp/data/pytabc.h5
```

❶ Stores the ran_int ndarray object.

❷ Stores the ran_flo ndarray object.

❸ The changes are reflected in the object description.

Working with Arrays2

- Writing these objects directly to an HDF5 database is faster than looping over the objects and writing the data row-by-row to a `Table` object or using the approach via structured `ndarray` objects.

HDF5 is a data model, library, and file format for storing and managing data. It supports an unlimited variety of datatypes, and is designed for flexible and efficient I/O and for high volume and complex data. HDF5 is portable and is extensible, allowing applications to evolve in their use of HDF5. The HDF5 Technology suite includes tools and applications for managing, manipulating, viewing, and analyzing data in the HDF5 format.

HDF5-BASED DATA STORAGE

The HDF5 hierarchical database (file) format is a powerful alternative to, for example, relational databases when it comes to structured numerical and financial data. Both on a standalone basis when using `PyTables` directly and when combining it with the capabilities of `pandas`, one can expect to get almost the maximum I/O performance that the available hardware allows.

https://portal.hdfgroup.org/display/HDF5/HDF5

Out of Memory Computations1

- `PyTables` supports out-of-memory operations, which makes it possible to implement array-based computations that do not fit in memory.
- To this end, consider the following code based on the `EArray` class.
- This type of object can be expanded in one dimension (row-wise), while the number of columns (elements per row) needs to be fixed:

❶ The fixed number of columns.

❷ The path and technical name of the `EArray` object.

❸ The atomic `dtype` object of the single values.

❹ The shape for instantiation (no rows, `n` columns).

❺ The `ndarray` object with the random numbers …

❻ … that gets appended many times.

https://www.pytables.org/_modules/tables/earray.html

```
In [182]: filename = path + 'earray.h5'

In [183]: h5 = tb.open_file(filename, 'w')

In [184]: n = 500  ❶

In [185]: ear = h5.create_earray('/', 'ear',  ❷
                                 atom=tb.Float64Atom(),  ❸
                                 shape=(0, n))  ❹

In [186]: type(ear)
Out[186]: tables.earray.EArray

In [187]: rand = np.random.standard_normal((n, n))  ❺
          rand[:4, :4]
Out[187]: array([[-1.25983231,  1.11420699,  0.1667485 ,  0.7345676 ],
                 [-0.13785424,  1.22232417,  1.36303097,  0.13521042],
                 [ 1.45487119, -1.47784078,  0.15027672,  0.86755989],
                 [-0.63519366,  0.1516327 , -0.64939447, -0.45010975]])

In [188]: %%time
          for _ in range(750):
              ear.append(rand)  ❻
          ear.flush()
          CPU times: user 814 ms, sys: 1.18 s, total: 1.99 s
          Wall time: 2.53 s

In [189]: ear
Out[189]: /ear (EArray(375000, 500)) ''
            atom := Float64Atom(shape=(), dflt=0.0)
            maindim := 0
            flavor := 'numpy'
            byteorder := 'little'
            chunkshape := (16, 500)

In [190]: ear.size_on_disk
Out[190]: 1500032000
```

Out of Memory Computations2

- For out-of-memory computations that do not lead to aggregations, another `EArray` object of the same shape (size) is needed. `PyTables` has a special module to cope with numerical expressions efficiently.
- It is called `Expr` and is based on the numerical expression library `numexpr`.
- The code that follows uses `Expr` to calculate the mathematical expression in Equation 9-1 on the whole `EArray` object from before.

Equation 9-1. Example mathematical expression

$$y = 3\sin(x) + \sqrt{|x|}$$

Out of Memory Computations3

- The results are stored in the `out` `EArray` object, and the expression evaluation happens chunk-wise:

❶ Transforms a `str` object–based expression to an `Expr` object.

❷ Defines the output to be the `out` `EArray` object.

❸ Initiates the evaluation of the expression.

❹ Reads the whole `EArray` into memory.

```
In [191]: out = h5.create_earray('/', 'out',
                                 atom=tb.Float64Atom(),
                                 shape=(0, n))

In [192]: out.size_on_disk
Out[192]: 0

In [193]: expr = tb.Expr('3 * sin(ear) + sqrt(abs(ear))')  ❶

In [194]: expr.set_output(out, append_mode=True)  ❷

In [195]: %time expr.eval()  ❸
          CPU times: user 3.08 s, sys: 1.7 s, total: 4.78 s
          Wall time: 4.03 s

Out[195]: /out (EArray(375000, 500)) ''
            atom := Float64Atom(shape=(), dflt=0.0)
            maindim := 0
            flavor := 'numpy'
            byteorder := 'little'
            chunkshape := (16, 500)

In [196]: out.size_on_disk
Out[196]: 1500032000

In [197]: out[0, :10]
Out[197]: array([-1.73369462,  3.74824436,  0.90627898,  2.86786818,
           1.75424957,
          -0.91108973, -1.68313885,  1.29073295, -1.68665599, -1.71345309])

In [198]: %time out_ = out.read()  ❹
          CPU times: user 1.03 s, sys: 1.1 s, total: 2.13 s
          Wall time: 2.22 s

In [199]: out_[0, :10]
Out[199]: array([-1.73369462,  3.74824436,  0.90627898,  2.86786818,
           1.75424957,
          -0.91108973, -1.68313885,  1.29073295, -1.68665599, -1.71345309])
```

Out of Memory Computations4

- Given that the whole operation takes place out-of-memory, it can be considered quite fast, in particular as it is executed on standard hardware.
- As a benchmark, the in-memory performance of the `numexpr` module (see also Chapter 10) can be considered.
- It is faster, but not by a huge margin:

```
In [200]: import numexpr as ne ❶

In [201]: expr = '3 * sin(out_) + sqrt(abs(out_))' ❷

In [202]: ne.set_num_threads(1) ❸
Out[202]: 4

In [203]: %time ne.evaluate(expr)[0, :10] ❹
          CPU times: user 2.51 s, sys: 1.54 s, total: 4.05 s
          Wall time: 4.94 s

Out[203]: array([-1.64358578,  0.22567882,  3.31363043,  2.50443549,
           4.27413965,
          -1.41600606, -1.68373023,  4.01921805, -1.68117412, -1.66053597])

In [204]: ne.set_num_threads(4) ❺
Out[204]: 1

In [205]: %time ne.evaluate(expr)[0, :10] ❻
          CPU times: user 3.39 s, sys: 1.94 s, total: 5.32 s
          Wall time: 2.96 s

Out[205]: array([-1.64358578,  0.22567882,  3.31363043,  2.50443549,
           4.27413965,
          -1.41600606, -1.68373023,  4.01921805, -1.68117412, -1.66053597])

In [206]: h5.close()
```

❶ Imports the module for *in-memory* evaluations of numerical expressions.

❷ The numerical expression as a `str` object.

❸ Sets the number of threads to one.

❹ Evaluates the numerical expression in-memory with one thread.

❺ Sets the number of threads to four.

❻ Evaluates the numerical expression in-memory with four threads.

https://numexpr.readthedocs.io/projects/NumExpr3/en/latest/api.html?highlight=set_num_thread#numexpr.set_num_threads

`numexpr.set_num_threads`(*nthreads*) [source]

Sets a number of threads to be used in operations.

Returns the previous setting for the number of threads.

During initialization time Numexpr sets this number to the number of detected cores in the system (see *detect_number_of_cores()*).

If you are using Intel's VML, you may want to use *set_vml_num_threads(nthreads)* to perform the parallel job with VML instead. However, you should get very similar performance with VML-optimized functions, and VML's parallelizer cannot deal with common expresions like *(x+1)*(x-2)*, while Numexpr's one can.

I/O with TsTables

- The package `TsTables` uses `PyTables` to build a high-performance storage for time series data.
- The major usage scenario is “write once, retrieve multiple times.”
- This is a typical scenario in financial analytics, where data is created in the markets, retrieved in real-time or asynchronously, and stored on disk for later usage.
- That usage might be in a larger trading strategy backtesting program that requires different subsets of a historical financial time series over and over again.
- It is then important that data retrieval happens fast.

Sample Data1

- As usual, the first task is the generation of a sample data set that is large enough to illustrate the benefits of `TsTables`.
- The following code generates three rather long financial time series based on the simulation of a geometric Brownian motion.


```
In [208]: no = 5000000  ❶
          co = 3  ❷
          interval = 1. / (12 * 30 * 24 * 60)  ❸
          vol = 0.2  ❹

In [209]: %%time
          rn = np.random.standard_normal((no, co))  ❺
          rn[0] = 0.0  ❻
          paths = 100 * np.exp(np.cumsum(-0.5 * vol ** 2 * interval +
                  vol * np.sqrt(interval) * rn, axis=0))  ❼
          paths[0] = 100  ❽
          CPU times: user 869 ms, sys: 175 ms, total: 1.04 s
          Wall time: 812 ms
```

❶ The number of time steps.

❷ The number of time series.

❸ The time interval as a year fraction.

❹ The volatility.

❺ Standard normally distributed random numbers.

❻ Sets the initial random numbers to zero.

❼ The simulation based on an Euler discretization.

❽ Sets the initial values of the paths to 100.

See next slide & Chapter 12

Simulation: Chapter 12: Dynamically simulated geometric Brownian motion paths

- Monte Carlo simulation (MCS) is among the most important numerical techniques in finance, if not the most important and widely used.
- This mainly stems from the fact that it is the most flexible numerical method when it comes to the evaluation of mathematical expressions (e.g., integrals), and specifically the valuation of financial derivatives.
- The flexibility comes at the cost of a relatively high computational burden, though, since often hundreds of thousands or even millions of complex computations have to be carried out to come up with a single value estimate.
- https://www.ibm.com/uk-en/cloud/learn/monte-carlo-simulation
- https://www.investopedia.com/terms/m/montecarlosimulation.asp

Random Variables

Consider, for example, the Black-Scholes-Merton setup for option pricing. In their setup, the level of a stock index S_T at a future date T given a level S_0 as of today is given according to Equation 12-1.

Equation 12-1. Simulating future index level in Black-Scholes-Merton setup

$$S_T = S_0 \exp\left(\left(r - \frac{1}{2}\sigma^2\right)T + \sigma\sqrt{T}z\right)$$

The variables and parameters have the following meaning:

S_T
: Index level at date T

r
: Constant riskless short rate

σ
: Constant volatility (= standard deviation of returns) of S

z
: Standard normally distributed random variable

Sample Data2

- Since `TsTables` works pretty well with `pandas DataFrame` objects, the data is transformed to such an object (see also Figure 9-7):

```
In [210]: dr = pd.date_range('2019-1-1', periods=no, freq='1s')

In [211]: dr[-6:]
Out[211]: DatetimeIndex(['2019-02-27 20:53:14', '2019-02-27 20:53:15',
                         '2019-02-27 20:53:16', '2019-02-27 20:53:17',
                         '2019-02-27 20:53:18', '2019-02-27 20:53:19'],
                        dtype='datetime64[ns]', freq='S')

In [212]: df = pd.DataFrame(paths, index=dr, columns=['ts1', 'ts2', 'ts3'])

In [213]: df.info()
          <class 'pandas.core.frame.DataFrame'>
          DatetimeIndex: 5000000 entries, 2019-01-01 00:00:00 to 2019-02-27
           20:53:19
          Freq: S
          Data columns (total 3 columns):
          ts1      float64
          ts2      float64
          ts3      float64
          dtypes: float64(3)
          memory usage: 152.6 MB

In [214]: df.head()
Out[214]:
                                  ts1         ts2         ts3
          2019-01-01 00:00:00  100.000000  100.000000  100.000000
          2019-01-01 00:00:01  100.018443   99.966644   99.998255
          2019-01-01 00:00:02  100.069023  100.004420   99.986646
          2019-01-01 00:00:03  100.086757  100.000246   99.992042
          2019-01-01 00:00:04  100.105448  100.036033   99.950618

In [215]: df[::100000].plot(figsize=(10, 6));
```

Sample Data3: Dynamically simulated geometric Brownian motion paths

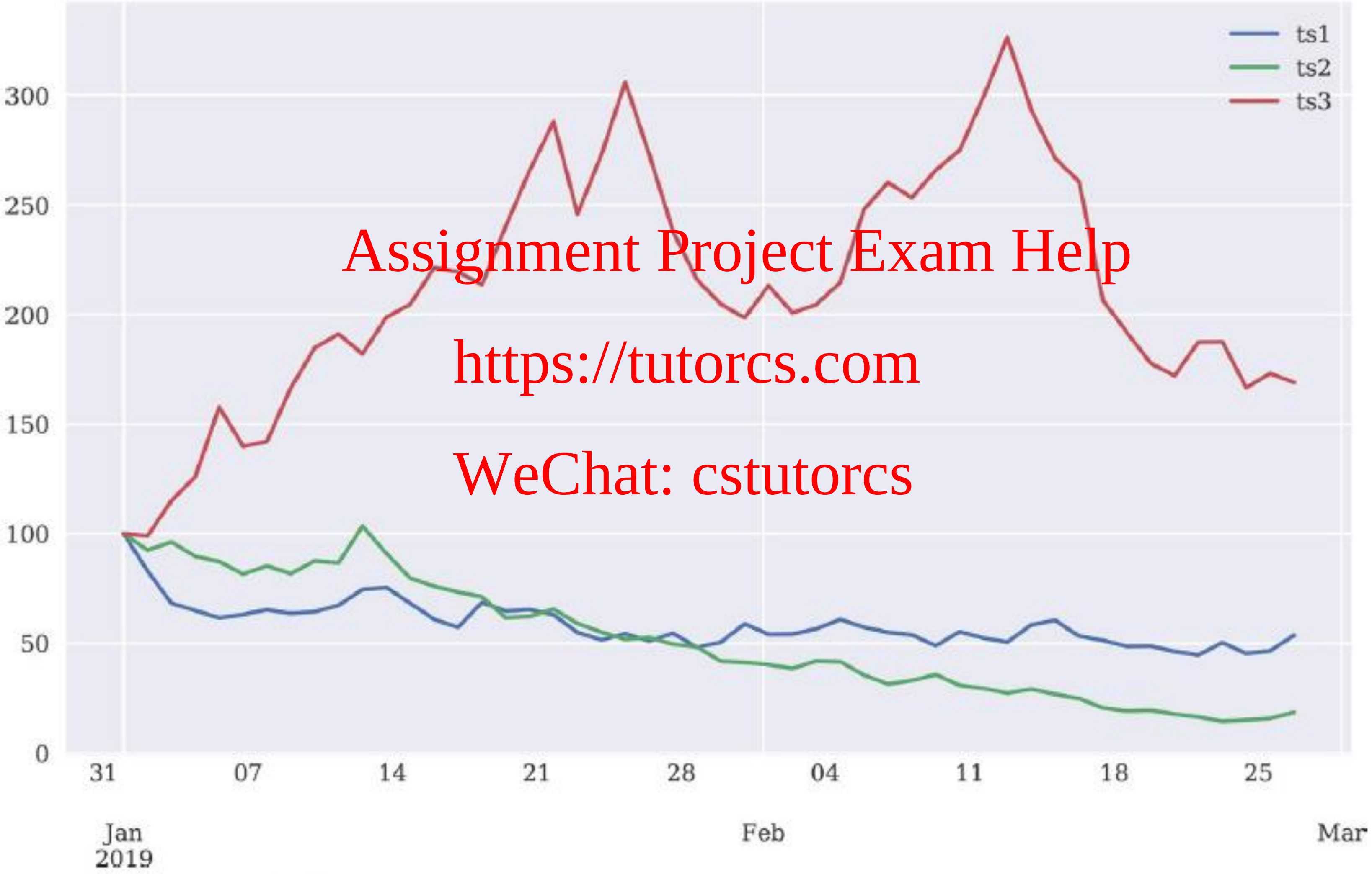


Figure 9-7. Selected data points of the financial time series

Data Storage

- `TsTables` stores financial time series data based on a specific chunk-based structure that allows for fast retrieval of arbitrary data subsets defined by some time interval.
- To this end, the package adds the function `create_ts()` to `PyTables`.
- To provide the data types for the table columns, the following uses a method based on the `tb.IsDe scription` class from `PyTables`:


```
In [216]: import tstables as tstab

In [217]: class ts_desc(tb.IsDescription):
              timestamp = tb.Int64Col(pos=0)  ❶
              ts1 = tb.Float64Col(pos=1)  ❷
              ts2 = tb.Float64Col(pos=2)  ❷
              ts3 = tb.Float64Col(pos=3)  ❷

In [218]: h5 = tb.open_file(path + 'tstab.h5', 'w')  ❸

In [219]: ts = h5.create_ts('/', 'ts', ts_desc)  ❹

In [220]: %time ts.append(df)  ❺
          CPU times: user 1.36 s, sys: 497 ms, total: 1.86 s
          Wall time: 1.29 s

In [221]: type(ts)
Out[221]: tstables.tstable.TsTable
```

❶ The column for the timestamps.

❷ The columns to store the numerical data.

❸ Opens an HDF5 database file for writing (`w`).

❹ Creates the `TsTable` object based on the `ts_desc` object.

❺ Appends the data from the `DataFrame` object to the `TsTable` objec

Data Retrieval1

- Writing data with `TsTables` obviously is quite fast, even if hardware dependent.
- The same holds true for reading chunks of the data back into memory.
- Conveniently, `TsTables` returns a `DataFrame` object (see also Figure 9-8):

```
In [223]: read_start_dt = dt.datetime(2019, 2, 1, 0, 0)  ❶
          read_end_dt = dt.datetime(2019, 2, 5, 23, 59)  ❷

In [224]: %time rows = ts.read_range(read_start_dt, read_end_dt)  ❸
          CPU times: user 182 ms, sys: 73.5 ms, total: 255 ms
          Wall time: 163 ms

In [225]: rows.info()  ❹
          <class 'pandas.core.frame.DataFrame'>
          DatetimeIndex: 431941 entries, 2019-02-01 00:00:00 to 2019-02-0
           23:59:00
          Data columns (total 3 columns):
          ts1    431941 non-null float64
          ts2    431941 non-null float64
          ts3    431941 non-null float64
          dtypes: float64(3)
          memory usage: 13.2 MB

In [226]: rows.head()  ❹
Out[226]:                            ts1        ts2         ts3
          2019-02-01 00:00:00  52.063640  40.474580  217.324713
          2019-02-01 00:00:01  52.087455  40.471911  217.250070
          2019-02-01 00:00:02  52.084808  40.458013  217.228712
          2019-02-01 00:00:03  52.073536  40.451408  217.302912
          2019-02-01 00:00:04  52.056133  40.450951  217.207481

In [227]: h5.close()

In [228]: (rows[::500] / rows.iloc[0]).plot(figsize=(10, 6));
```


❶ The start time of the interval.

❷ The end time of the interval.

❸ The function `ts.read_range()` returns a `DataFrame` object for the interval.

❹ The `DataFrame` object has a few hundred thousand data rows.

Data Retrieval2

Figure 9-8. A specific time interval of the financial time series (normalized)

Data Retrieval3

- To better illustrate the performance of the `TsTables`-based data retrieval, consider the following benchmark, which retrieves 100 chunks of data consisting of 3 days' worth of 1-second bars.
- The retrieval of a `DataFrame` with 345,600 rows of data takes less than one-tenth of a second:

```
In [229]: import random

In [230]: h5 = tb.open_file(path + 'tstab.h5', 'r')

In [231]: ts = h5.root.ts._f_get_timeseries()  ❶

In [232]: %%time
          for _ in range(100):  ❷
              d = random.randint(1, 24)  ❸
              read_start_dt = dt.datetime(2019, 2, d, 0, 0, 0)
              read_end_dt = dt.datetime(2019, 2, d + 3, 23, 59, 59)
              rows = ts.read_range(read_start_dt, read_end_dt)
          CPU times: user 7.17 s, sys: 1.65 s, total: 8.81 s
          Wall time: 4.18 s

In [233]: rows.info()  ❹

          <class 'pandas.core.frame.DataFrame'>
          DatetimeIndex: 345600 entries, 2019-02-04 00:00:00 to 2019-02-07
           23:59:59
          Data columns (total 3 columns):
          ts1    345600 non-null float64
          ts2    345600 non-null float64
          ts3    345600 non-null float64
          dtypes: float64(3)
          memory usage: 10.5 MB
```


❶ This connects to the `TsTable` object.

❷ The data retrieval is repeated many times.

❸ The starting day value is randomized.

❹ The last `DataFrame` object is retrieved.

Conclusion

- SQL-based or relational databases have advantages when it comes to complex data structures that exhibit lots of relations between single objects/tables.
- This might justify in some circumstances their performance disadvantage over pure `NumPy ndarray`-based or `pandas DataFrame`–based approaches.
- Many application areas in finance or science in general can succeed with a mainly array-based data modelling approach.
- In these cases, huge performance improvements can be realized by making use of native `NumPy` I/O capabilities, a combination of `NumPy` and `PyTables` capabilities, or the `pandas` approach via HDF5-based stores.
- `TsTables` is particularly useful when working with large (financial) time series data sets, especially in "write once, retrieve multiple times" scenarios.
- While a recent trend has been to use cloud-based solutions — where the cloud is made up of a large number of computing nodes based on commodity hardware — one should carefully consider, especially in a financial context, which hardware architecture best serves the analytics requirements.
- Companies, research institutions, and others involved in data analytics should therefore analyze first what specific tasks have to be accomplished in general and then decide on the hardware/software architecture, in terms of:

Scaling out

- Using a cluster with many commodity nodes with standard CPUs and relatively low memory

Scaling up

- Using one or a few powerful servers with many-core CPUs, possibly also GPUs or even TPUs when machine and deep learning play a role, and large amounts of memory

Further Resources

- The paper cited at the beginning of the chapter is a good read, and a good starting point to think about hardware architecture for financial analytics:
  - Appuswamy, Raja, et al. (2013). "Nobody Ever Got Fired for Buying a Cluster". Microsoft Technical Report.
- As usual, the web provides many valuable resources with regard to the topics and Python packages covered in this chapter:

- For serialization of Python objects with `pickle`, refer to the documentation.
  https://docs.python.org/3/library/pickle.html

- An overview of the I/O capabilities of `NumPy` is provided on the website.
  https://numpy.org/doc/stable/reference/routines.io.html

- For I/O with `pandas`, see the respective section in the online documentation.
  https://pandas.pydata.org/pandas-docs/stable/user_guide/io.html
- The `PyTables` home page provides both tutorials and detailed documentation. http://www.pytables.org/#
- More information on `TsTables` can be found on its GitHub page. https://github.com/afiedler/tstables/
- A friendly fork for `TsTables` is found at *http://github.com/yhilpisch/tstables*.
- Use `pip install git+git://github.com/yhilpisch/tstables` to install the package from this fork, which is maintained for compatibility with newer versions of `pandas` and other Python packages.