Pandas

BASIC TO ADVANCED

FOR MACHINE LEARNING

karan

KARAN SINGH BISHT
SR. DEV

COMPLETE
GUIDE TO
LEARN NUMPY

Pandas as Pd

From basic to advance

FOR MACHINE LEARNING

-Karan Singh Bisht

ABOUT THE BOOK

Learn how to use pandas, a flexible and fast Python toolkit for data manipulation, analysis, and discovery.

About This Book* Get comfortable using pandas and Python as a data exploration and analysis tool

*** Explore pandas through a data analysis framework, with an explanation of how pandas is well suited for the various stages in the data analysis process**

*** A comprehensive guide to pandas, with many clear and practical examples to help you get up and running with pandas This book is great for data scientists, data analysts, Python programmers who want to dive into data analysis with pandas, and anybody with an interest in data analysis.**

Some knowledge of statistics and programming will be helpful to get the most out of this book but not strictly required. Prior exposure to pandas is also not required.What You Will Learn

* Understand how data analysts and scientists think about of the processes of gathering and understanding data

* Learn how pandas can be used to support the end-to-end process of data analysis

* Use pandas Series and DataFrame objects to represent single and multivariate data

* Slicing and dicing data with pandas, as well as combining, grouping, and aggregating data from multiple sources

* How to access data from external sources such as files, databases, and web services

* Represent and manipulate time-series data and the many of the intricacies involved with this type of data* How to visualize statistical information

* How to use pandas to solve several common data representation and analysis problems within financeIn DetailYou will learn how to use pandas to perform data analysis in Python. You will start with an overview of data analysis and iteratively progress from modeling data, to accessing data from remote sources, performing numeric and statistical analysis, through indexing and performing aggregate analysis, and finally to visualizing statistical data and applying pandas to finance.With the knowledge you gain from this book, you will quickly learn pandas and how it can empower you in the exciting world of data manipulation, analysis and science.Style and approach

* Step-by-step instruction on using pandas within an end-to-end framework of performing data analysis

* Practical demonstration of using Python and pandas using interactive and incremental examples

SUMMARY

OUTLINE
Pandas as Pd
Pandas Introduction

Pandas Getting Started
Pandas Series
Labels
Create Labels
Pandas DataFrames
What is a DataFrame?
Locate Row
Named Indexes
Locate Named Indexes
DataFrame
From dict of Series or dicts
From a Series
From a list of namedtuples
From a list of dataclasses

Alternate constructors
DataFrame.from_records

Column selection, addition, deletion

Assigning new columns in method chains
Indexing / selection
Data alignment and arithmetic

Operations with scalars are just as you would expect:
Boolean operators work as well:
Transposing

DataFrame interoperability with NumPy functions
Console display
DataFrame column attribute access and IPython completion
Read CSV Files
IO tools (text, CSV, HDF5, …)
CSV & text files
Parsing options
Basic
Column and index locations and names
usecolslist-like or callable, default None
squeezeboolean, default False
prefixstr, default None

General parsing configuration
convertersdict, default None

skipfooterint, default 0
memory_mapboolean, default False

NA and missing data handling
keep_default_naboolean, default True
na_filterboolean, default True
Datetime handling
infer_datetime_formatboolean, default False
keep_date_colboolean, default False
date_parserfunction, default None
dayfirstboolean, default False
cache_datesboolean, default True
Iteration
chunksizeint, default None
Quoting, compression, and file format
Thousands separator.
quotecharstr (length 1)
doublequoteboolean, default True
Error handling
Specifying column data types

dtype: object

Specifying categorical dtype

Naming and using columns
Handling column names

Duplicate names parsing

Filtering columns (usecols)
Comments and empty lines
Ignoring line comments and empty lines

Comments
Dealing with Unicode data
Index columns and trailing delimiters
Date Handling
Specifying date columns
Date parsing functions
Parsing a CSV with mixed timezones
Inferring datetime format
International date formats
Writing CSVs to binary file objects
Specifying method for floating-point conversion
Thousand separators
The thousands keyword allows integers to be parsed correctly:
NA values
Above, only an empty field will be recognized as NaN.

Above, both NA and 0 as strings are NaN.

Infinity

Returning Series

Boolean values

Handling “bad” lines

Dialect

Quoting and Escape Characters

Files with fixed width columns

Indexes

Files with an “implicit” index column

Reading an index with a MultiIndex

Reading columns with a MultiIndex

Automatically “sniffing” the delimiter

Reading multiple files to create a single DataFrame

Iterating through files chunk by chunk

Specifying the parser engine

Reading/writing remote files

Writing out data

Writing to CSV format

Writing a formatted string

JSON

Writing JSON

Orient options

Date handling

Writing in ISO date format, with microseconds:

Epoch timestamps, in seconds:

Writing to a file, with a date index and a date column:

Fallback behavior

Reading JSON

Data conversion

Dates written in nanoseconds need to be read back in nanoseconds:

# Try to parse timestamps as milliseconds -> Won't Work

The Numpy parameter

Normalization

Line delimited json

# reader is an iterator that returns ``chunksize`` lines each iteration

Empty DataFrame

Table schema

HTML

Reading HTML content

Read a URL and match a table that contains specific text:

Specify an index column:

Specify a number of rows to skip:

Specify a number of rows to skip using a list (range works as well):
Specify an HTML attribute:
Specify values that should be converted to NaN:
Specify whether to keep the default set of NaN values:
url_mcc,
Read in pandas to_html output (with some loss of floating point precision):
Or you could pass flavor='lxml' without a list:
Writing to HTML files
HTML Table Parsing Gotchas
Issues with lxml
Data Cleaning
Our Data Set
Empty Cells
Remove Rows

Replace Empty Values
Replace Only For Specified Columns
Replace Using Mean, Median, or Mode
Pandas - Cleaning Data of Wrong Format
Data of Wrong Format
Convert Into a Correct Format
Removing Rows
Pandas - Fixing Wrong Data
Wrong Data
Replacing Values
Removing Rows
Pandas - Removing Duplicates
Discovering Duplicates
Removing Duplicates
Chart Visualization
Basic plotting: plot
Other plots
Bar plots
Histograms
Box plots
Area plot
Scatter plot
Hexagonal bin plot
Pie plot
Plotting with missing data
Plotting tools
Scatter matrix plot
Density plot
Andrews curves
Parallel coordinates
Lag plot

Autocorrelation plot
Bootstrap plot
RadViz
Plot formatting
Setting the plot style
General plot style arguments
Controlling the legend
Controlling the labels
Scales
Plotting on a secondary y-axis
Custom formatters for timeseries plots
Suppressing tick resolution adjustment
Automatic date tick adjustment
Subplots
Using layout and targeting multiple axes
Plotting with error bars
# Generate the data
# errors should be positive, and defined in the order of lower, upper
# Plot
Plotting tables
Colormaps
Plotting directly with matplotlib
Plotting backends
Scaling to large datasets
Load less data
Use efficient datatypes
Use chunking
Use other libraries
Enhancing performance
Note
Cython (writing C extensions for pandas)
Pure Python
Plain Cython
Adding type
Using ndarray
So, do not do this:
More advanced techniques
1000 loops, best of 3: 987 us per loop
Numba (JIT compilation)
pandas Numba Engine
# Run the first time, compilation time will affect performance
188 ms ± 1.93 ms per loop (mean ± std. dev. of 7 runs, 10 loops each)
Caveats
Expression evaluation via eval()
Note
Note

Supported syntax
Statements
8.34 ms +- 147 us per loop (mean +- std. dev. of 7 runs, 100 loops each)
7.47 ms +- 199 us per loop (mean +- std. dev. of 7 runs, 100 loops each)
23.8 ms +- 697 us per loop (mean +- std. dev. of 7 runs, 10 loops each)
The DataFrame.eval method
Local variables
pandas.eval() parsers
pandas.eval() backends
Note
8.31 ms +- 544 us per loop (mean +- std. dev. of 7 runs, 100 loops each)
pandas.eval() performance
Technical minutia regarding expression evaluation
Python For Data Science Cheat Sheet: Pandas Basics
import pandas as pd
Pandas Data Structures
Series
s = pd.Series([3, -5, 7, 4], index=['a', 'b', 'c', 'd'])
DataFrame
Asking For Help
help(pd.Series.loc)
I/O
Read and Write to CSV
Read multiple sheets from the same file
Read and Write to Excel
Read and Write to SQL Query or Database Table
*pd.read_sql(SELECT * FROM my_table;, engine)*
*pd.read_sql_query(SELECT * FROM my_table;', engine)*
Selection
Getting
Get subset of a DataFrame
Country Capital Population
Selecting', Boolean Indexing and Setting
By Positio'
By Label/Position
Select a single column of subset of columns
Select rows and columns
'New Delhi'
Boolean Indexing
s[~(s > 1)]
s where value is <-1 or >2
s[(s < -1) | (s > 2)]
Use filter to adjust DataFrame
df[df['Population']>1200000000]
Setting
s['a'] = 6

Dropping
s.drop(['a', 'c'])
Drop values from columns(axis=1)
df.drop('Country', axis=1)
Sort and Rank
df.sort_index()
Sort by the values along an axis
df.sort_values(by='Country')
Assign ranks to entries
df.rank()
Retrieving Series/DataFrame Information
Basic Information
df.shape
Describe index
df.index
Describe DataFrame columns
df.columns
Info on DataFrame
df.info()
Number of non-NA values
df.count()
Summary
df.sum()
Cumulative sum of values
df.cumsum()
Minimum/maximum values
df.min()/df.max()
Minimum/Maximum index value
Summary statistics
Mean of values
Median of value
Apply function
Apply function element-wise
df.applynap(f)
Internal Data Alignment
Arithmetic Operations with Fill Methods

Pandas Introduction

What is Pandas?

Pandas is a Python library used for working with data sets.

It has functions for analyzing, cleaning, exploring, and manipulating data.

The name "Pandas" has a reference to both "Panel Data", and "Python Data Analysis" and was created by Wes McKinney in 2008.

Why Use Pandas?

Pandas allows us to analyze big data and make conclusions based on statistical theories.

Pandas can clean messy data sets, and make them readable and relevant.

Relevant data is very important in data science.

Data Science: is a branch of computer science where we study how to store, use and analyze data for deriving information from it.

What Can Pandas Do?

Pandas gives you answers about the data. Like:

- Is there a correlation between two or more columns?
- What is average value?
- Max value?
- Min value?

Pandas are also able to delete rows that are not relevant, or contains wrong values, like empty or NULL values. This is called *cleaning* the data.

Where is the Pandas Codebase?

The source code for Pandas is located at this github repository https://github.com/pandas-dev/pandas

github: enables many people to work on the same codebase.

Pandas Getting Started

Installation of Pandas

If you have Python and PIP already installed on a system, then installation of Pandas is very easy.

Install it using this command:

```
C:\Users\Your Name>pip install pandas
```

If this command fails, then use a python distribution that already has Pandas installed like, Anaconda, Spyder etc.

Import Pandas

Once Pandas is installed, import it in your applications by adding the import keyword:

```
import pandas
```

Now Pandas is imported and ready to use.

Example

```
import pandas
mydataset = {
  'cars': ["BMW", "Volvo", "Ford"],
  'passings': [3, 7, 2]
}
myvar = pandas.DataFrame(mydataset)
print(myvar)
```

Pandas Series

What is a Series?

A Pandas Series is like a column in a table.

It is a one-dimensional array holding data of any type.

Example

Create a simple Pandas Series from a list:

```
import pandas as pd

a = [1, 7, 2]

myvar = pd.Series(a)

print(myvar)
```

Labels

If nothing else is specified, the values are labeled with their index number. First value has index 0, second value has index 1 etc.

This label can be used to access a specified value.

Example

Return the first value of the Series:

```
print(myvar[0])
```

Create Labels

With the index argument, you can name your own labels.

Example

Create your own labels:

```
import pandas as pd

a = [1, 7, 2]

myvar = pd.Series(a, index = ["x", "y", "z"])

print(myvar)
```

When you have created labels, you can access an item by referring to the label.

Example

Return the value of "y":

```
print(myvar["y"])
```

Pandas DataFrames

What is a DataFrame?

A Pandas DataFrame is a 2 dimensional data structure, like a 2 dimensional array, or a table with rows and columns.

Example

Create a simple Pandas DataFrame:

```
import pandas as pd

data = {

  "calories": [420, 380, 390],

  "duration": [50, 40, 45]

}

#load data into a DataFrame object:

df = pd.DataFrame(data)
```

```
print(df)
```

Locate Row

As you can see from the result above, the DataFrame is like a table with rows and columns.

Pandas use the loc attribute to return one or more specified row(s)

Example

Return row 0:

```
#refer to the row index:

print(df.loc[0])
```

```
 calories    420
duration     50
Name: 0, dtype: int64
```

Note: This example returns a Pandas Series.

Example

Return row 0 and 1:

```
#use a list of indexes:
```

```
print(df.loc[[0, 1]])
```

Result

```
  calories  duration
0       420        50
1       380        40
```

Note: When using [], the result is a Pandas DataFrame.

Named Indexes

With the index argument, you can name your own indexes.

Example

Add a list of names to give each row a name:

```
import pandas as pd

data = {

  "calories": [420, 380, 390],

  "duration": [50, 40, 45]

}


df = pd.DataFrame(data, index = ["day1", "day2", "day3"])

print(df)
```

Result

```
      calories  duration
day1       420        50
day2       380        40
day3       390        45
```

Locate Named Indexes

Use the named index in the loc attribute to return the specified row(s).

Example

Return "day2":

```
#refer to the named index:

print(df.loc["day2"])
```

Result

```
calories    380
duration     40
Name: 0, dtype: int64
```

Load Files Into a DataFrame

If your data sets are stored in a file, Pandas can load them into a DataFrame.

Example

Load a comma separated file (CSV file) into a DataFrame:

```
import pandas as pd

df = pd.read_csv('data.csv')

print(df)
```

You will learn more about importing files in the next chapters.

Series

Series is a one-dimensional labeled array capable of holding any data type (integers, strings, floating point numbers, Python objects, etc.). The axis labels are collectively referred to as the index. The basic method to create a Series is to call:

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

Here, data can be many different things:

- a Python dict
- an ndarray
- a scalar value (like 5)

The passed index is a list of axis labels. Thus, this separates into a few cases depending on what data is:

From ndarray

If data is an ndarray, index must be the same length as data. If no index is passed, one will be created having values [0, ..., len(data) - 1].

```
In [3]: s = pd.Series(np.random.randn(5), index=["a", "b", "c", "d", "e"])

In [4]: s
Out[4]:
a    0.469112
b   -0.282863
```

```
c   -1.509059
d   -1.135632
e    1.212112
dtype: float64

In [5]: s.index
Out[5]: Index(['a', 'b', 'c', 'd', 'e'], dtype='object')

In [6]: pd.Series(np.random.randn(5))
Out[6]:
0   -0.173215
1    0.119209
2   -1.044236
3   -0.861849
4   -2.104569
dtype: float64
```

Note

pandas supports non-unique index values. If an operation that does not support duplicate index values is attempted, an exception will be raised at that time. The reason for being lazy is nearly all performance-based (there are many instances in computations, like parts of GroupBy, where the index is not used).

From dict

Series can be instantiated from dicts:

```
In [7]: d = {"b": 1, "a": 0, "c": 2}

In [8]: pd.Series(d)
Out[8]:
b    1
```

```
a    0
c    2
dtype: int64
```

Note

When the data is a dict, and an index is not passed, the Series index will be ordered by the dict's insertion order, if you're using Python version >= 3.6 and pandas version >= 0.23.

If you're using Python < 3.6 or pandas < 0.23, and an index is not passed, the Series index will be the lexically ordered list of dict keys.

In the example above, if you were on a Python version lower than 3.6 or a pandas version lower than 0.23, the Series would be ordered by the lexical order of the dict keys (i.e. ['a', 'b', 'c'] rather than ['b', 'a', 'c']).

If an index is passed, the values in data corresponding to the labels in the index will be pulled out.

```
In [9]: d = {"a": 0.0, "b": 1.0, "c": 2.0}

In [10]: pd.Series(d)
Out[10]:
a    0.0
b    1.0
c    2.0
dtype: float64

In [11]: pd.Series(d, index=["b", "c", "d", "a"])
Out[11]:
b    1.0
c    2.0
d    NaN
a    0.0
dtype: float64
```

Note

NaN (not a number) is the standard missing data marker used in pandas.

From scalar value

If data is a scalar value, an index must be provided. The value will be repeated to match the length of index.

```
In [12]: pd.Series(5.0, index=["a", "b", "c", "d", "e"])
Out[12]:
a    5.0
b    5.0
c    5.0
d    5.0
e    5.0
dtype: float64
```

Series is ndarray-like

Series acts very similarly to a ndarray, and is a valid argument to most NumPy functions. However, operations such as slicing will also slice the index.

```
In [13]: s[0]
Out[13]: 0.4691122999071863

In [14]: s[:3]
Out[14]:
a    0.469112
b   -0.282863
c   -1.509059
dtype: float64
```

```
In [15]: s[s > s.median()]
Out[15]:
a    0.469112
e    1.212112
dtype: float64

In [16]: s[[4, 3, 1]]
Out[16]:
e    1.212112
d   -1.135632
b   -0.282863
dtype: float64

In [17]: np.exp(s)
Out[17]:
a    1.598575
b    0.753623
c    0.221118
d    0.321219
e    3.360575
dtype: float64
```

Note

We will address array-based indexing like s[[4, 3, 1]] in section on indexing.

Like a NumPy array, a pandas Series has a dtype.

```
In [18]: s.dtype
Out[18]: dtype('float64')
```

This is often a NumPy dtype. However, pandas and 3rd-party libraries extend NumPy’s type system in a few places, in which case the dtype would be an ExtensionDtype. Some examples within pandas are Categorical data and Nullable integer data type. See dtypes for more.

If you need the actual array backing a Series, use Series.array.

```
In [19]: s.array
Out[19]:
<PandasArray>
[ 0.4691122999071863, -0.2828633443286633, -1.5090585031735124,
-1.1356323710171934,  1.2121120250208506]
Length: 5, dtype: float64
```

Accessing the array can be useful when you need to do some operation without the index (to disable automatic alignment, for example).

Series.array will always be an ExtensionArray. Briefly, an ExtensionArray is a thin wrapper around one or more *concrete* arrays like a numpy.ndarray. pandas knows how to take an ExtensionArray and store it in a Series or a column of a DataFrame. See dtypes for more.

While Series is ndarray-like, if you need an *actual* ndarray, then use Series.to_numpy().

```
In [20]: s.to_numpy()
Out[20]: array([ 0.4691, -0.2829, -1.5091, -1.1356,  1.2121])
```

Even if the Series is backed by a ExtensionArray, Series.to_numpy() will return a NumPy ndarray.

Series is dict-like

A Series is like a fixed-size dict in that you can get and set values by index label:

```
In [21]: s["a"]
Out[21]: 0.4691122999071863

In [22]: s["e"] = 12.0
In [23]: s
Out[23]:
a     0.469112
b    -0.282863
c    -1.509059
d    -1.135632
e    12.000000
dtype: float64

In [24]: "e" in s
Out[24]: True

In [25]: "f" in s
Out[25]: False
```

If a label is not contained, an exception is raised:

```
>>> s["f"]
KeyError: 'f'
```

Using the get method, a missing label will return None or specified default:

```
In [26]: s.get("f")

In [27]: s.get("f", np.nan)
Out[27]: nan
```

Vectorized operations and label alignment with Series

When working with raw NumPy arrays, looping through value-by-value is usually not necessary. The same is true when working with Series in pandas. Series can also be passed into most NumPy methods expecting an ndarray.

```
In [28]: s + s
```

```
Out[28]:
a     0.938225
b    -0.565727
c    -3.018117
d    -2.271265
e    24.000000
dtype: float64

In [29]: s * 2
Out[29]:
a     0.938225
b    -0.565727
c    -3.018117
d    -2.271265
e    24.000000
dtype: float64

In [30]: np.exp(s)
Out[30]:
a         1.598575
b         0.753623
c         0.221118
d         0.321219
e    162754.791419
dtype: float64
```

A key difference between Series and ndarray is that operations between Series automatically align the data based on label. Thus, you can write computations without giving consideration to whether the Series involved have the same labels.

```
In [31]: s[1:] + s[:-1]
Out[31]:
a         NaN
b   -0.565727
c   -3.018117
```

```
d   -2.271265
e         NaN
dtype: float64
```

The result of an operation between unaligned Series will have the union of the indexes involved. If a label is not found in one Series or the other, the result will be marked as missing NaN. Being able to write code without doing any explicit data alignment grants immense freedom and flexibility in interactive data analysis and research. The integrated data alignment features of the pandas data structures set pandas apart from the majority of related tools for working with labeled data.

Note

In general, we chose to make the default result of operations between differently indexed objects yield the union of the indexes in order to avoid loss of information. Having an index label, though the data is missing, is typically important information as part of a computation. You of course have the option of dropping labels with missing data via the dropna function.

Name attribute

Series can also have a name attribute:

```
In [32]: s = pd.Series(np.random.randn(5), name="something")

In [33]: s
Out[33]:
0   -0.494929
1    1.071804
2    0.721555
3   -0.706771
4   -1.039575
Name: something, dtype: float64

In [34]: s.name
Out[34]: 'something'
```

The Series name will be assigned automatically in many cases, in particular when taking 1D slices of DataFrame as you will see below.

You can rename a Series with the pandas.Series.rename() method.

```
In [35]: s2 = s.rename("different")

In [36]: s2.name
Out[36]: 'different'
```

Note that s and s2 refer to different objects.

DataFrame

DataFrame is a 2-dimensional labeled data structure with columns of potentially different types. You can think of it like a spreadsheet or SQL table, or a dict of Series objects. It is generally the most commonly used pandas object. Like Series, DataFrame accepts many different kinds of input:

- Dict of 1D ndarrays, lists, dicts, or Series
- 2-D numpy.ndarray
- Structured or record ndarray
- A Series
- Another DataFrame

Along with the data, you can optionally pass index (row labels) and columns (column labels) arguments. If you pass an index and / or columns, you are guaranteeing the index and / or columns of the resulting DataFrame. Thus, a dict of Series plus a specific index will discard all data not matching up to the passed index.

If axis labels are not passed, they will be constructed from the input data based on common sense rules.

Note

When the data is a dict, and columns is not specified, the DataFrame columns will be ordered by the dict’s insertion order, if you are using Python version >= 3.6 and pandas >= 0.23.

If you are using Python < 3.6 or pandas < 0.23, and columns is not specified, the DataFrame columns will be the lexically ordered list of dict keys.

From dict of Series or dicts

The resulting index will be the union of the indexes of the various Series. If there are any nested dicts, these will first be converted to Series. If no columns are passed, the columns will be the ordered list of dict keys.

```
In [37]: d = {
  ....:     "one": pd.Series([1.0, 2.0, 3.0], index=["a", "b", "c"]),
  ....:     "two": pd.Series([1.0, 2.0, 3.0, 4.0], index=["a", "b", "c", "d"]),
  ....: }
  ....:

In [38]: df = pd.DataFrame(d)
In [39]: df
Out[39]:
   one  two
a  1.0  1.0
b  2.0  2.0
c  3.0  3.0
d  NaN  4.0

In [40]: pd.DataFrame(d, index=["d", "b", "a"])
Out[40]:
   one  two
d  NaN  4.0
b  2.0  2.0
a  1.0  1.0

In [41]: pd.DataFrame(d, index=["d", "b", "a"], columns=["two", "three"])
Out[41]:
   two three
```

```
d  4.0   NaN
b  2.0   NaN
a  1.0   NaN
```

The row and column labels can be accessed respectively by accessing the index and columns attributes:

Note

When a particular set of columns is passed along with a dict of data, the passed columns override the keys in the dict.

```
In [42]: df.index
Out[42]: Index(['a', 'b', 'c', 'd'], dtype='object')

In [43]: df.columns
Out[43]: Index(['one', 'two'], dtype='object')
```

From dict of ndarrays / lists

The ndarrays must all be the same length. If an index is passed, it must clearly also be the same length as the arrays. If no index is passed, the result will be range(n), where n is the array length.

```
In [44]: d = {"one": [1.0, 2.0, 3.0, 4.0], "two": [4.0, 3.0, 2.0, 1.0]}

In [45]: pd.DataFrame(d)
Out[45]:
   one  two
0  1.0  4.0
1  2.0  3.0
2  3.0  2.0
3  4.0  1.0

In [46]: pd.DataFrame(d, index=["a", "b", "c", "d"])
Out[46]:
   one  two
a  1.0  4.0
b  2.0  3.0
c  3.0  2.0
d  4.0  1.0
```

From structured or record array

This case is handled identically to a dict of arrays.

```
In [47]: data = np.zeros((2,), dtype=[("A", "i4"), ("B", "f4"), ("C", "a10")])

In [48]: data[:] = [(1, 2.0, "Hello"), (2, 3.0, "World")]

In [49]: pd.DataFrame(data)
Out[49]:
   A    B         C
0  1  2.0  b'Hello'
1  2  3.0  b'World'

In [50]: pd.DataFrame(data, index=["first", "second"])
Out[50]:
        A    B         C
first   1  2.0  b'Hello'
second  2  3.0  b'World'

In [51]: pd.DataFrame(data, columns=["C", "A", "B"])
Out[51]:
          C  A    B
0  b'Hello'  1  2.0
1  b'World'  2  3.0
```

Note

DataFrame is not intended to work exactly like a **2-dimensional NumPy ndarray.**

From a list of dicts

```
In [52]: data2 = [{"a": 1, "b": 2}, {"a": 5, "b": 10, "c": 20}]

In [53]: pd.DataFrame(data2)
Out[53]:
   a   b     c
0  1   2   NaN
1  5  10  20.0

In [54]: pd.DataFrame(data2, index=["first", "second"])
Out[54]:
        a   b     c
first   1   2   NaN
second  5  10  20.0

In [55]: pd.DataFrame(data2, columns=["a", "b"])
Out[55]:
   a   b
0  1   2
1  5  10
```

From a dict of tuples

You can automatically create a MultiIndexed frame by passing a tuples dictionary.

```
In [56]: pd.DataFrame(
   ....:     {
   ....:         ("a", "b"): {("A", "B"): 1, ("A", "C"): 2},
   ....:         ("a", "a"): {("A", "C"): 3, ("A", "B"): 4},
   ....:         ("a", "c"): {("A", "B"): 5, ("A", "C"): 6},
   ....:         ("b", "a"): {("A", "C"): 7, ("A", "B"): 8},
   ....:         ("b", "b"): {("A", "D"): 9, ("A", "B"): 10},
   ....:     }
   ....: )
   ....: 
Out[56]: 
       a              b      
       b    a    c    a     b
A B  1.0  4.0  5.0  8.0  10.0
  C  2.0  3.0  6.0  7.0   NaN
  D  NaN  NaN  NaN  NaN   9.0
```

From a Series

The result will be a DataFrame with the same index as the input Series, and with one column whose name is the original name of the Series (only if no other column name provided).

From a list of namedtuples

The field names of the first namedtuple in the list determine the columns of the DataFrame. The remaining namedtuples (or tuples) are simply unpacked and their values are fed into the rows of the DataFrame. If any of those tuples is shorter than the first namedtuple then the later columns in the corresponding row are marked as missing values. If any are longer than the first namedtuple, a ValueError is raised.

```
In [57]: from collections import namedtuple

In [58]: Point = namedtuple("Point", "x y")

In [59]: pd.DataFrame([Point(0, 0), Point(0, 3), (2, 3)])
Out[59]: 
   x  y
0  0  0
1  0  3
2  2  3
```

```
In [60]: Point3D = namedtuple("Point3D", "x y z")
In [61]: pd.DataFrame([Point3D(0, 0, 0), Point3D(0, 3, 5), Point(2, 3)])
```

```
Out[61]:
   x  y    z
0  0  0  0.0
1  0  3  5.0
2  2  3  NaN
```

From a list of dataclasses

Data Classes , can be passed into the DataFrame constructor. Passing a list of dataclasses is equivalent to passing a list of dictionaries.

Please be aware, that all values in the list should be dataclasses, mixing types in the list would result in a TypeError.

```
In [62]: from dataclasses import make_dataclass

In [63]: Point = make_dataclass("Point", [("x", int), ("y", int)])

In [64]: pd.DataFrame([Point(0, 0), Point(0, 3), Point(2, 3)])
Out[64]:
   x  y
0  0  0
1  0  3
2  2  3
```

Missing data

Much more will be said on this topic in the Missing data section. To construct a DataFrame with missing data, we use np.nan to represent missing values. Alternatively, you may pass a numpy.MaskedArray as the data argument to the DataFrame constructor, and its masked entries will be considered missing.

Alternate constructors

DataFrame.from_dict

DataFrame.from_dict takes a dict of dicts or a dict of array-like sequences and returns a DataFrame. It operates like the DataFrame constructor except for the orient parameter which is 'columns' by default, but which can be set to 'index' in order to use the dict keys as row labels.

```
In [65]: pd.DataFrame.from_dict(dict([("A", [1, 2, 3]), ("B", [4, 5, 6])]))
Out[65]:
   A  B
0  1  4
1  2  5
2  3  6
```

If you pass orient='index', the keys will be the row labels. In this case, you can also pass the desired column names:

```
In [66]: pd.DataFrame.from_dict(
   ....:     dict([("A", [1, 2, 3]), ("B", [4, 5, 6])]),
   ....:     orient="index",
   ....:     columns=["one", "two", "three"],
   ....: )
   ....:
Out[66]:
   one  two  three
A    1    2      3
B    4    5      6
```

DataFrame.from_records

DataFrame.from_records takes a list of tuples or an ndarray with structured dtype. It works analogously to the normal DataFrame constructor, except that the resulting DataFrame index may be a specific field of the structured dtype. For example:

```
In [67]: data
Out[67]:
array([(1, 2., b'Hello'), (2, 3., b'World')],
     dtype=[('A', '<i4'), ('B', '<f4'), ('C', 'S10')])

In [68]: pd.DataFrame.from_records(data, index="C")
Out[68]:
          A    B
C
b'Hello'  1  2.0
b'World'  2  3.0
```

Column selection, addition, deletion

You can treat a DataFrame semantically like a dict of like-indexed Series objects. Getting, setting, and deleting columns works with the same syntax as the analogous dict operations:

```
In [69]: df["one"]
Out[69]:
a    1.0
b    2.0
c    3.0
d    NaN
Name: one, dtype: float64

In [70]: df["three"] = df["one"] * df["two"]
In [71]: df["flag"] = df["one"] > 2

In [72]: df
Out[72]:
   one  two  three   flag
a  1.0  1.0    1.0  False
b  2.0  2.0    4.0  False
c  3.0  3.0    9.0   True
d  NaN  4.0    NaN  False
```

Columns can be deleted or popped like with a dict:

```
In [73]: del df["two"]

In [74]: three = df.pop("three")

In [75]: df
Out[75]:
   one   flag
a  1.0  False
b  2.0  False
c  3.0   True
d  NaN  False
```

When inserting a scalar value, it will naturally be propagated to fill the column:

```
In [76]: df["foo"] = "bar"
In [77]: df
Out[77]:
   one   flag  foo
a  1.0  False  bar
b  2.0  False  bar
c  3.0   True  bar
```

```
d  NaN  False  bar
```

When inserting a Series that does not have the same index as the DataFrame, it will be conformed to the DataFrame's index:

```
In [78]: df["one_trunc"] = df["one"][:2]

In [79]: df
Out[79]:
   one   flag  foo  one_trunc
a  1.0  False  bar        1.0
b  2.0  False  bar        2.0
c  3.0   True  bar        NaN
d  NaN  False  bar        NaN
```

You can insert raw ndarrays but their length must match the length of the DataFrame's index.

By default, columns get inserted at the end. The insert function is available to insert at a particular location in the columns:

```
In [80]: df.insert(1, "bar", df["one"])

In [81]: df
Out[81]:
   one  bar   flag  foo  one_trunc
a  1.0  1.0  False  bar        1.0
b  2.0  2.0  False  bar        2.0
c  3.0  3.0   True  bar        NaN
d  NaN  NaN  False  bar        NaN
```

Assigning new columns in method chains

Inspired by dplyr's mutate verb, DataFrame has an assign() method that allows you to easily create new columns that are potentially derived from existing columns.

```
In [82]: iris = pd.read_csv("data/iris.data")

In [83]: iris.head()
Out[83]: 
   SepalLength  SepalWidth  PetalLength  PetalWidth         Name
0          5.1         3.5          1.4         0.2  Iris-setosa
1          4.9         3.0          1.4         0.2  Iris-setosa
2          4.7         3.2          1.3         0.2  Iris-setosa
3          4.6         3.1          1.5         0.2  Iris-setosa
4          5.0         3.6          1.4         0.2  Iris-setosa

In [84]: iris.assign(sepal_ratio=iris["SepalWidth"] / iris["SepalLength"]).head()
Out[84]: 
   SepalLength  SepalWidth  PetalLength  PetalWidth         Name  sepal_ratio
0          5.1         3.5          1.4         0.2  Iris-setosa     0.686275
1          4.9         3.0          1.4         0.2  Iris-setosa     0.612245
2          4.7         3.2          1.3         0.2  Iris-setosa     0.680851
3          4.6         3.1          1.5         0.2  Iris-setosa     0.673913
4          5.0         3.6          1.4         0.2  Iris-setosa     0.720000
```

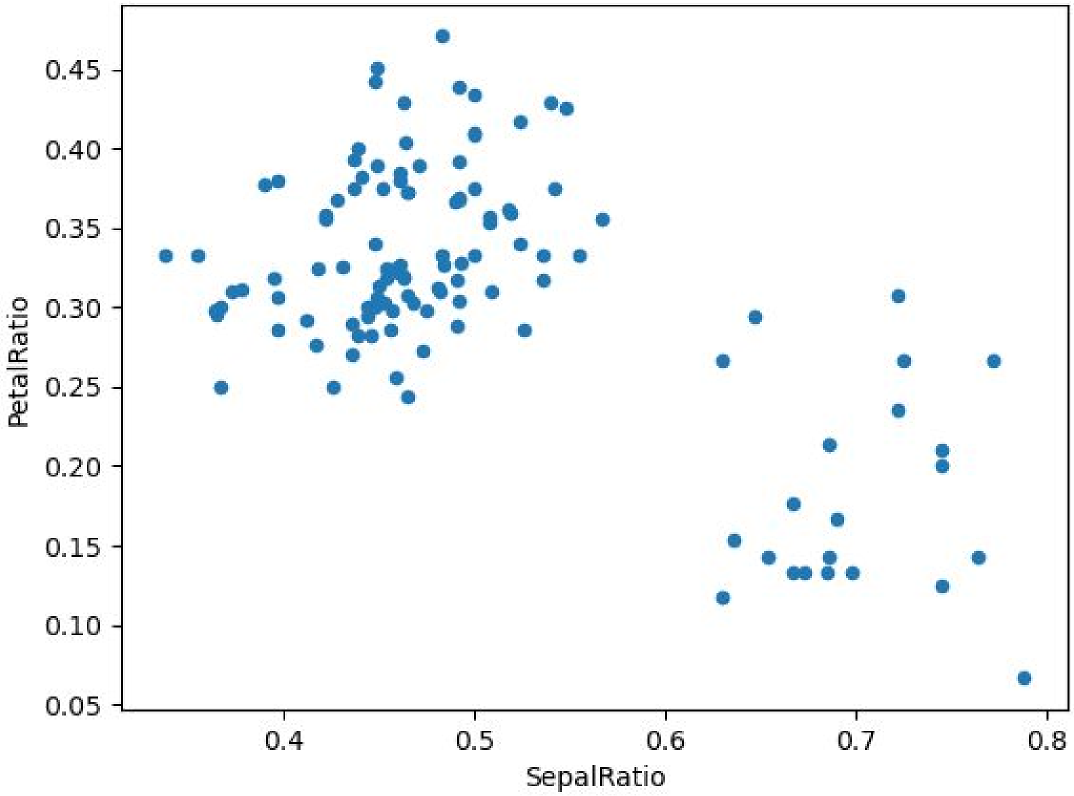
In the example above, we inserted a precomputed value. We can also pass in a function of one argument to be evaluated on the DataFrame being assigned to.

```
In [85]: iris.assign(sepal_ratio=lambda x: (x["SepalWidth"] / x["SepalLength"])).head()
Out[85]: 
   SepalLength  SepalWidth  PetalLength  PetalWidth         Name  sepal_ratio
0          5.1         3.5          1.4         0.2  Iris-setosa     0.686275
1          4.9         3.0          1.4         0.2  Iris-setosa     0.612245
2          4.7         3.2          1.3         0.2  Iris-setosa     0.680851
3          4.6         3.1          1.5         0.2  Iris-setosa     0.673913
4          5.0         3.6          1.4         0.2  Iris-setosa     0.720000
```

assign always returns a copy of the data, leaving the original DataFrame untouched.

Passing a callable, as opposed to an actual value to be inserted, is useful when you don’t have a reference to the DataFrame at hand. This is common when using assign in a chain of operations. For example, we can limit the DataFrame to just those observations with a Sepal Length greater than 5, calculate the ratio, and plot:

```
In [86]: (
   ....:     iris.query("SepalLength > 5")
   ....:     .assign(
   ....:         SepalRatio=lambda x: x.SepalWidth / x.SepalLength,
   ....:         PetalRatio=lambda x: x.PetalWidth / x.PetalLength,
   ....:     )
   ....:     .plot(kind="scatter", x="SepalRatio", y="PetalRatio")
   ....: )
   ....:
Out[86]: <AxesSubplot:xlabel='SepalRatio', ylabel='PetalRatio'>
```



Since a function is passed in, the function is computed on the DataFrame being assigned to. Importantly, this is the DataFrame that's been filtered to those rows with sepal length greater than 5. The filtering happens first, and then the ratio calculations. This is an example where we didn't have a reference to the *filtered* DataFrame available.

The function signature for assign is simply **kwargs. The keys are the column names for the new fields, and the values are either a value to be inserted (for example, a Series or NumPy array), or a function of one argument to be called on the DataFrame. A *copy* of the original DataFrame is returned, with the new values inserted.

Starting with Python 3.6 the order of **kwargs is preserved. This allows for *dependent* assignment, where an expression later in **kwargs can refer to a column created earlier in the same assign().

```
In [87]: dfa = pd.DataFrame({"A": [1, 2, 3], "B": [4, 5, 6]})
In [88]: dfa.assign(C=lambda x: x["A"] + x["B"], D=lambda x: x["A"] +
x["C"])
Out[88]:
   A  B  C   D
0  1  4  5   6
1  2  5  7   9
2  3  6  9  12
```

In the second expression, x['C'] will refer to the newly created column, that's equal to dfa['A'] + dfa['B'].

Indexing / selection

The basics of indexing are as follows:

Operation	Syntax	Result
Select column	df[col]	Series
Select row by label	df.loc[label]	Series
Select row by integer location	df.iloc[loc]	Series
Slice rows	df[5:10]	DataFrame
Select rows by boolean vector	df[bool_vec]	DataFrame

Row selection, for example, returns a Series whose index is the columns of the DataFrame:

```
In [89]: df.loc["b"]
Out[89]:
one        2.0
bar        2.0
flag     False
```

```
foo            bar
one_trunc      2.0
Name: b, dtype: object

In [90]: df.iloc[2]
Out[90]:
one          3.0
bar          3.0
flag        True
foo          bar
one_trunc    NaN
Name: c, dtype: object
```

For a more exhaustive treatment of sophisticated label-based indexing and slicing, see the section on indexing. We will address the fundamentals of reindexing / conforming to new sets of labels in the section on reindexing.

Data alignment and arithmetic

Data alignment between DataFrame objects automatically align on both the columns and the index (row labels). Again, the resulting object will have the union of the column and row labels.

```
In [91]: df = pd.DataFrame(np.random.randn(10, 4), columns=["A", "B", "C",
"D"])

In [92]: df2 = pd.DataFrame(np.random.randn(7, 3), columns=["A", "B", "C"])

In [93]: df + df2
Out[93]:
          A         B         C   D
0  0.045691 -0.014138  1.380871 NaN
1 -0.955398 -1.501007  0.037181 NaN
2 -0.662690  1.534833 -0.859691 NaN
3 -2.452949  1.237274 -0.133712 NaN
4  1.414490  1.951676 -2.320422 NaN
5 -0.494922 -1.649727 -1.084601 NaN
```

```
6 -1.047551 -0.748572 -0.805479 NaN
7       NaN       NaN       NaN NaN
8       NaN       NaN       NaN NaN
9       NaN       NaN       NaN NaN
```

When doing an operation between DataFrame and Series, the default behavior is to align the Series index on the DataFrame columns, thus broadcasting row-wise. For example:

```
In [94]: df - df.iloc[0]
Out[94]:
          A         B         C         D
0  0.000000  0.000000  0.000000  0.000000
1 -1.359261 -0.248717 -0.453372 -1.754659
2  0.253128  0.829678  0.010026 -1.991234
3 -1.311128  0.054325 -1.724913 -1.620544
4  0.573025  1.500742 -0.676070  1.367331
5 -1.741248  0.781993 -1.241620 -2.053136
6 -1.240774 -0.869551 -0.153282  0.000430
7 -0.743894  0.411013 -0.929563 -0.282386
8 -1.194921  1.320690  0.238224 -1.482644
9  2.293786  1.856228  0.773289 -1.446531
```

For explicit control over the matching and broadcasting behavior, see the section on flexible binary operations.

Operations with scalars are just as you would expect:

```
In [95]: df * 5 + 2
Out[95]:
          A         B         C          D
0  3.359299 -0.124862  4.835102   3.381160
1 -3.437003 -1.368449  2.568242  -5.392133
2  4.624938  4.023526  4.885230  -6.575010
3 -3.196342  0.146766 -3.789461  -4.721559
4  6.224426  7.378849  1.454750  10.217815
```

```
5  -5.346940  3.785103 -1.373001  -6.884519
6  -2.844569 -4.472618  4.068691   3.383309
7  -0.360173  1.930201  0.187285   1.969232
8  -2.615303  6.478587  6.026220  -4.032059
9  14.828230  9.156280  8.701544  -3.851494

In [96]: 1 / df
Out[96]:
          A          B         C           D
0  3.678365  -2.353094  1.763605    3.620145
1 -0.919624  -1.484363  8.799067   -0.676395
2  1.904807   2.470934  1.732964   -0.583090
3 -0.962215  -2.697986 -0.863638   -0.743875
4  1.183593   0.929567 -9.170108    0.608434
5 -0.680555   2.800959 -1.482360   -0.562777
6 -1.032084  -0.772485  2.416988    3.614523
7 -2.118489 -71.634509 -2.758294 -162.507295
8 -1.083352   1.116424  1.241860   -0.828904
9  0.389765   0.698687  0.746097   -0.854483

In [97]: df ** 4
Out[97]:
           A             B         C             D
0   0.005462  3.261689e-02  0.103370  5.822320e-03
1   1.398165  2.059869e-01  0.000167  4.777482e+00
2   0.075962  2.682596e-02  0.110877  8.650845e+00
3   1.166571  1.887302e-02  1.797515  3.265879e+00
4   0.509555  1.339298e+00  0.000141  7.297019e+00
5   4.661717  1.624699e-02  0.207103  9.969092e+00
6   0.881334  2.808277e+00  0.029302  5.858632e-03
7   0.049647  3.797614e-08  0.017276  1.433866e-09
8   0.725974  6.437005e-01  0.420446  2.118275e+00
9  43.329821  4.196326e+00  3.227153  1.875802e+00
```

Boolean operators work as well:

```
In [98]: df1 = pd.DataFrame({"a": [1, 0, 1], "b": [0, 1, 1]}, dtype=bool)

In [99]: df2 = pd.DataFrame({"a": [0, 1, 1], "b": [1, 1, 0]}, dtype=bool)

In [100]: df1 & df2
Out[100]:
       a      b
0  False  False
1  False   True
2   True  False

In [101]: df1 | df2
Out[101]:
      a     b
0  True  True
1  True  True
2  True  True

In [102]: df1 ^ df2
Out[102]:
       a      b
0   True   True
1   True  False
2  False   True

In [103]: -df1
Out[103]:
       a      b
0  False   True
1   True  False
2  False  False
```

Transposing

To transpose, access the T attribute (also the transpose function), similar to an ndarray:

```
# only show the first 5 rows
In [104]: df[:5].T
Out[104]:
          0         1         2         3         4
A  0.271860 -1.087401  0.524988 -1.039268  0.844885
B -0.424972 -0.673690  0.404705 -0.370647  1.075770
C  0.567020  0.113648  0.577046 -1.157892 -0.109050
D  0.276232 -1.478427 -1.715002 -1.344312  1.643563
```

DataFrame interoperability with NumPy functions

Elementwise NumPy ufuncs (log, exp, sqrt, …) and various other NumPy functions can be used with no issues on Series and DataFrame, assuming the data within are numeric:

```
In [105]: np.exp(df)
Out[105]:
           A         B         C         D
0   1.312403  0.653788  1.763006  1.318154
1   0.337092  0.509824  1.120358  0.227996
2   1.690438  1.498861  1.780770  0.179963
3   0.353713  0.690288  0.314148  0.260719
4   2.327710  2.932249  0.896686  5.173571
5   0.230066  1.429065  0.509360  0.169161
6   0.379495  0.274028  1.512461  1.318720
7   0.623732  0.986137  0.695904  0.993865
8   0.397301  2.449092  2.237242  0.299269
9  13.009059  4.183951  3.820223  0.310274

In [106]: np.asarray(df)
Out[106]:
array([[ 0.2719, -0.425 ,  0.567 ,  0.2762],
    [-1.0874, -0.6737,  0.1136, -1.4784],
    [ 0.525 ,  0.4047,  0.577 , -1.715 ],
    [-1.0393, -0.3706, -1.1579, -1.3443],
    [ 0.8449,  1.0758, -0.109 ,  1.6436],
    [-1.4694,  0.357 , -0.6746, -1.7769],
    [-0.9689, -1.2945,  0.4137,  0.2767],
    [-0.472 , -0.014 , -0.3625, -0.0062],
    [-0.9231,  0.8957,  0.8052, -1.2064],
    [ 2.5656,  1.4313,  1.3403, -1.1703]])
```

DataFrame is not intended to be a drop-in replacement for ndarray as its indexing semantics and data model are quite different in places from an n-dimensional array.

Series implements __array_ufunc__, which allows it to work with NumPy’s universal functions.

The ufunc is applied to the underlying array in a Series.

```
In [107]: ser = pd.Series([1, 2, 3, 4])

In [108]: np.exp(ser)
Out[108]:
0     2.718282
1     7.389056
2    20.085537
3    54.598150
dtype: float64
```

Changed in version 0.25.0: When multiple Series are passed to a ufunc, they are aligned before performing the operation.

Like other parts of the library, pandas will automatically align labeled inputs as part of a ufunc with multiple inputs. For example, using numpy.remainder() on two Series with differently ordered labels will align before the operation.

```
In [109]: ser1 = pd.Series([1, 2, 3], index=["a", "b", "c"])

In [110]: ser2 = pd.Series([1, 3, 5], index=["b", "a", "c"])

In [111]: ser1
Out[111]:
a    1
b    2
c    3
dtype: int64

In [112]: ser2
Out[112]:
b    1
a    3
c    5
dtype: int64

In [113]: np.remainder(ser1, ser2)
Out[113]:
a    1
b    0
c    3
dtype: int64
```

As usual, the union of the two indices is taken, and non-overlapping values are filled with missing values.

```
In [114]: ser3 = pd.Series([2, 4, 6], index=["b", "c", "d"])

In [115]: ser3
Out[115]:
b    2
c    4
d    6
dtype: int64
```

```
In [116]: np.remainder(ser1, ser3)
Out[116]:
a    NaN
b    0.0
c    3.0
d    NaN
dtype: float64
```

When a binary ufunc is applied to a Series and Index, the Series implementation takes precedence and a Series is returned.

```
In [117]: ser = pd.Series([1, 2, 3])

In [118]: idx = pd.Index([4, 5, 6])

In [119]: np.maximum(ser, idx)
Out[119]:
0    4
1    5
2    6
dtype: int64
```

NumPy ufuncs are safe to apply to Series backed by non-ndarray arrays, for example arrays.SparseArray (see Sparse calculation). If possible, the ufunc is applied without converting the underlying data to an ndarray.

Console display

Very large DataFrames will be truncated to display them in the console. You can also get a summary using info(). (Here I am reading a CSV version of the baseball dataset from the plyr R package):

```
In [120]: baseball = pd.read_csv("data/baseball.csv")

In [121]: print(baseball)
```

```
      id     player  year  stint team  lg   g   ab   r    h  X2b  X3b  hr   rbi   sb   cs
bb    so  ibb  hbp   sh   sf  gidp
0   88641  womacto01  2006     2  CHN  NL  19   50   6   14    1    0   1   2.0
1.0  1.0   4   4.0  0.0  0.0  3.0  0.0   0.0
1   88643  schilcu01  2006     1  BOS  AL  31    2   0    1    0    0   0   0.0  0.0
0.0   0   1.0  0.0  0.0  0.0  0.0   0.0
..    ...        ...   ...    ...  ...  ..  ..  ...  ..  ...  ...  ...  ..   ...  ...  ...  ..   ...  ...  ...  ...  ...
...
98  89533   aloumo01  2007     1  NYN  NL  87  328  51  112   19    1  13
49.0  3.0  0.0  27  30.0  5.0  2.0  0.0  3.0  13.0
99  89534  alomasa02  2007     1  NYN  NL   8   22   1    3    1    0   0   0.0
0.0  0.0   0   3.0  0.0  0.0  0.0  0.0   0.0

[100 rows x 23 columns]

In [122]: baseball.info()
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 100 entries, 0 to 99
Data columns (total 23 columns):
#   Column  Non-Null Count  Dtype
---  ------  --------------  -----
0   id      100 non-null    int64
1   player  100 non-null    object
2   year    100 non-null    int64
3   stint   100 non-null    int64
4   team    100 non-null    object
5   lg      100 non-null    object
6   g       100 non-null    int64
7   ab      100 non-null    int64
8   r       100 non-null    int64
9   h       100 non-null    int64
10  X2b     100 non-null    int64
11  X3b     100 non-null    int64
12  hr      100 non-null    int64
13  rbi     100 non-null    float64
14  sb      100 non-null    float64
```

```
15  cs      100 non-null    float64
16  bb      100 non-null    int64
17  so      100 non-null    float64
18  ibb     100 non-null    float64
19  hbp     100 non-null    float64
20  sh      100 non-null    float64
21  sf      100 non-null    float64
22  gidp    100 non-null    float64
dtypes: float64(9), int64(11), object(3)
memory usage: 18.1+ KB
```

However, using to_string will return a string representation of the DataFrame in tabular form, though it won't always fit the console width:

```
In [123]: print(baseball.iloc[-20:, :12].to_string())
       id     player  year  stint team  lg    g   ab   r    h  X2b  X3b
80  89474   finlest01  2007      1  COL  NL   43   94   9   17    3    0
81  89480  embreal01  2007      1  OAK  AL    4    0   0    0    0    0
82  89481  edmonji01  2007      1  SLN  NL  117  365  39   92   15    2
83  89482  easleda01  2007      1  NYN  NL   76  193  24   54    6    0
84  89489  delgaca01  2007      1  NYN  NL  139  538  71  139   30    0
85  89493  cormirh01  2007      1  CIN  NL    6    0   0    0    0    0
86  89494  coninje01  2007      2  NYN  NL   21   41   2    8    2    0
87  89495  coninje01  2007      1  CIN  NL   80  215  23   57   11    1
88  89497  clemero02  2007      1  NYA  AL    2    2   0    1    0    0
89  89498  claytro01  2007      2  BOS  AL    8    6   1    0    0    0
90  89499  claytro01  2007      1  TOR  AL   69  189  23   48   14    0
91  89501  cirilje01  2007      2  ARI  NL   28   40   6    8    4    0
92  89502  cirilje01  2007      1  MIN  AL   50  153  18   40    9    2
93  89521  bondsba01  2007      1  SFN  NL  126  340  75   94   14    0
94  89523  biggicr01  2007      1  HOU  NL  141  517  68  130   31    3
95  89525  benitar01  2007      2  FLO  NL   34    0   0    0    0    0
96  89526  benitar01  2007      1  SFN  NL   19    0   0    0    0    0
97  89530  ausmubr01  2007      1  HOU  NL  117  349  38   82   16    3
98  89533   aloumo01  2007      1  NYN  NL   87  328  51  112   19    1
99  89534  alomasa02  2007      1  NYN  NL    8   22   1    3    1    0
```

Wide DataFrames will be printed across multiple rows by default:

```
In [124]: pd.DataFrame(np.random.randn(3, 12))
Out[124]:
         0         1         2         3         4         5         6         7         8         9 
10        11
0 -1.226825  0.769804 -1.281247 -0.727707 -0.121306 -0.097883  0.695775
0.341734  0.959726 -1.110336 -0.619976  0.149748
1 -0.732339  0.687738  0.176444  0.403310 -0.154951  0.301624 -2.179861
-1.369849 -0.954208  1.462696 -1.743161 -0.826591
2 -0.345352  1.314232  0.690579  0.995761  2.396780  0.014871  3.357427
-0.317441 -1.236269  0.896171 -0.487602 -0.082240
```

You can change how much to print on a single row by setting the display.width option:

```
In [125]: pd.set_option("display.width", 40)  # default is 80

In [126]: pd.DataFrame(np.random.randn(3, 12))
Out[126]:
         0         1         2         3         4         5         6         7         8         9 
10        11
0 -2.182937  0.380396  0.084844  0.432390  1.519970 -0.493662  0.600178
0.274230  0.132885 -0.023688  2.410179  1.450520
1  0.206053 -0.251905 -2.213588  1.063327  1.266143  0.299368 -0.863838
0.408204 -1.048089 -0.025747 -0.988387  0.094055
2  1.262731  1.289997  0.082423 -0.055758  0.536580 -0.489682  0.369374
-0.034571 -2.484478 -0.281461  0.030711  0.109121
```

You can adjust the max width of the individual columns by setting display.max_colwidth

```
In [127]: datafile = {
   .....:     "filename": ["filename_01", "filename_02"],
   .....:     "path": [
   .....:         "media/user_name/storage/folder_01/filename_01",
   .....:         "media/user_name/storage/folder_02/filename_02",
   .....:     ],
   .....: }
   .....:

In [128]: pd.set_option("display.max_colwidth", 30)

In [129]: pd.DataFrame(datafile)
Out[129]:
      filename                           path
0  filename_01  media/user_name/storage/fo...
1  filename_02  media/user_name/storage/fo...

In [130]: pd.set_option("display.max_colwidth", 100)

In [131]: pd.DataFrame(datafile)
Out[131]:
      filename                                           path
0  filename_01  media/user_name/storage/folder_01/filename_01
1  filename_02  media/user_name/storage/folder_02/filename_02
```

You can also disable this feature via the expand_frame_repr option. This will print the table in one block.

DataFrame column attribute access and IPython completion

If a DataFrame column label is a valid Python variable name, the column can be accessed like an attribute:

```
In [132]: df = pd.DataFrame({"foo1": np.random.randn(5), "foo2": np.random.randn(5)})

In [133]: df
Out[133]:
       foo1      foo2
```

```
0  1.126203  0.781836
1 -0.977349 -1.071357
2  1.474071  0.441153
3 -0.064034  2.353925
4 -1.282782  0.583787

In [134]: df.foo1
Out[134]:
0    1.126203
1   -0.977349
2    1.474071
3   -0.064034
4   -1.282782
Name: foo1, dtype: float64
```

The columns are also connected to the IPython completion mechanism so they can be tab-completed:

```
In [5]: df.foo<TAB>  # noqa: E225, E999
df.foo1  df.foo2
```

Read CSV Files

A simple way to store big data sets is to use CSV files (comma separated files).

CSV files contains plain text and is a well know format that can be read by everyone including Pandas.

In our examples we will be using a CSV file called 'data.csv'.

Download data.csv. or Open data.csv

Example

Load the CSV into a DataFrame:

```
import pandas as pd

df = pd.read_csv('data.csv')

print(df.to_string())
```

Tip: use to_string() to print the entire DataFrame.

If you have a large DataFrame with many rows, Pandas will only return the first 5 rows, and the last 5 rows:

Example

Print the DataFrame without the to_string() method:

```
import pandas as pd

df = pd.read_csv('data.csv')

print(df)
```

max_rows

The number of rows returned is defined in Pandas option settings.

You can check your system's maximum rows with the pd.options.display.max_rows statement.

Example

Check the number of maximum returned rows:

```
import pandas as pd
```

```
print(pd.options.display.max_rows)
```

In my system the number is 60, which means that if the DataFrame contains more than 60 rows, the print(df) statement will return only the headers and the first and last 5 rows.

You can change the maximum rows number with the same statement.

Example

Increase the maximum number of rows to display the entire DataFrame:

```
import pandas as pd

pd.options.display.max_rows = 9999

df = pd.read_csv('data.csv')

print(df)
```

IO tools (text, CSV, HDF5, …)

The pandas I/O API is a set of top level reader functions accessed like pandas.read_csv() that generally return a pandas object. The corresponding writer functions are object methods that are accessed like DataFrame.to_csv(). Below is a table containing available readers and writers.

Format Type	Data Description	Reader	Writer
text	CSV	read_csv	to_csv
text	Fixed-Width Text File	read_fwf	

text	JSON	read_json	to_json
text	HTML	read_html	to_html
text	LaTeX		Styler.to_latex
text	XML	read_xml	to_xml
text	Local clipboard	read_clipboard	to_clipboard
binary	MS Excel	read_excel	to_excel
binary	OpenDocument	read_excel	
binary	HDF5 Format	read_hdf	to_hdf
binary	Feather Format	read_feather	to_feather
binary	Parquet Format	read_parquet	to_parquet
binary	ORC Format	read_orc	
binary	Stata	read_stata	to_stata
binary	SAS	read_sas	
binary	SPSS	read_spss	
binary	Python Pickle Format	read_pickle	to_pickle
SQL	SQL	read_sql	to_sql

SQL	Google BigQuery	read_gbq	to_gbq

Here is an informal performance comparison for some of these IO methods.

Note

For examples that use the StringIO class, make sure you import it with from io import StringIO for Python 3.

CSV & text files

The workhorse function for reading text files (a.k.a. flat files) is read_csv(). See the cookbook for some advanced strategies.

Parsing options

read_csv() accepts the following common arguments:

Basic filepath_or_buffervarious

Either a path to a file (a str, pathlib.Path, or py:py._path.local.LocalPath), URL (including http, ftp, and S3 locations), or any object with a read() method (such as an open file or StringIO).

sepstr, defaults to ',' for read_csv(), \t for read_table()

Delimiter to use. If sep is None, the C engine cannot automatically detect the separator, but the Python parsing engine can, meaning the latter will be used and automatically detect the separator by Python’s builtin sniffer tool, csv.Sniffer. In addition, separators longer than 1 character and different from '\s+' will be interpreted as regular expressions and will also force the use of the Python parsing engine. Note that regex delimiters are prone to ignoring quoted data. Regex example: '\\r\\t'.

delimiterstr, default None

Alternative argument name for sep.

delim_whitespaceboolean, default False

Specifies whether or not whitespace (e.g. ' ' or '\t') will be used as the delimiter. Equivalent to setting sep='\s+'. If this option is set to True, nothing should be passed in for the delimiter parameter.

Column and index locations and names

headerint or list of ints, default 'infer'

Row number(s) to use as the column names, and the start of the data. Default behavior is to infer the column names: if no names are passed the behavior is identical to header=0 and column names are inferred from the first line of the file, if column names are passed explicitly then the behavior is identical to header=None. Explicitly pass header=0 to be able to replace existing names.

The header can be a list of ints that specify row locations for a MultiIndex on the columns e.g. [0,1,3]. Intervening rows that are not specified will be skipped (e.g. 2 in this example is skipped). Note that this parameter ignores commented lines and empty lines if skip_blank_lines=True, so header=0 denotes the first line of data rather than the first line of the file.

namesarray-like, default None

List of column names to use. If file contains no header row, then you should explicitly pass header=None. Duplicates in this list are not allowed.

index_colint, str, sequence of int / str, or False, optional, default None

Column(s) to use as the row labels of the DataFrame, either given as string name or column index. If a sequence of int / str is given, a MultiIndex is used.

Note: index_col=False can be used to force pandas to *not* use the first column as the index, e.g. when you have a malformed file with delimiters at the end of each line.

The default value of None instructs pandas to guess. If the number of fields in the column header row is equal to the number of fields in the body of the data file, then a default index is used. If it is larger, then the first columns are used as index so that the remaining number of fields in the body are equal to the number of fields in the header.

The first row after the header is used to determine the number of columns, which will go into the index. If the subsequent rows contain less columns than the first row, they are filled with NaN.

This can be avoided through usecols. This ensures that the columns are taken as is and the trailing data are ignored.

usecolslist-like or callable, default None

Return a subset of the columns. If list-like, all elements must either be positional (i.e. integer indices into the document columns) or strings that correspond to column names provided either by the user in names or inferred from the document header row(s). If names are given, the document header row(s) are not taken into account. For example, a valid list-like usecols parameter would be [0, 1, 2] or ['foo', 'bar', 'baz'].

Element order is ignored, so usecols=[0, 1] is the same as [1, 0]. To instantiate a DataFrame from data with element order preserved use pd.read_csv(data, usecols=['foo', 'bar'])[['foo', 'bar']] for columns in ['foo', 'bar'] order or pd.read_csv(data, usecols=['foo', 'bar'])[['bar', 'foo']] for ['bar', 'foo'] order.

If callable, the callable function will be evaluated against the column names, returning names where the callable function evaluates to True:

```
In [1]: import pandas as pd

In [2]: from io import StringIO

In [3]: data = "col1,col2,col3\na,b,1\na,b,2\nc,d,3"
In [4]: pd.read_csv(StringIO(data))
```

```
Out[4]:

  col1 col2  col3

0    a    b     1

1    a    b     2

2    c    d     3
```

```
In [5]: pd.read_csv(StringIO(data), usecols=lambda x: x.upper() in ["COL1",
"COL3"])
```

```
Out[5]:

  col1  col3

0    a     1

1    a     2

2    c     3
```

Using this parameter results in much faster parsing time and lower memory usage when using the c engine. The Python engine loads the data first before deciding which columns to drop.

squeezeboolean, default False

If the parsed data only contains one column then return a Series.

Deprecated since version 1.4.0: Append .squeeze("columns") to the call to {func_name} to squeeze the data.

prefixstr, default None

Prefix to add to column numbers when no header, e.g. ‘X’ for X0, X1, …

Deprecated since version 1.4.0: Use a list comprehension on the DataFrame’s columns after calling read_csv.

```
In [6]: data = "col1,col2,col3\na,b,1"

In [7]: df = pd.read_csv(StringIO(data)

In [8]: df.columns = [f"pre_{col}" for col in df.columns]

In [9]: df

Out[9]:

  pre_col1 pre_col2  pre_col3

0       a       b        1
```

mangle_dupe_colsboolean, default True

Duplicate columns will be specified as ‘X’, ‘X.1’…’X.N’, rather than ‘X’…’X’. Passing in False will cause data to be overwritten if there are duplicate names in the columns.

General parsing configuration

dtypeType name or dict of column -> type, default None

Data type for data or columns. E.g. {'a': np.float64, 'b': np.int32} (unsupported with engine='python'). Use str or object together with suitable na_values settings to preserve and not interpret dtype.

engine{'c', 'python', 'pyarrow'}

Parser engine to use. The C and pyarrow engines are faster, while the python engine is currently more feature-complete. Multithreading is currently only supported by the pyarrow engine.

New in version 1.4.0: The “pyarrow” engine was added as an *experimental* engine, and some features are unsupported, or may not work correctly, with this engine.

convertersdict, default None

Dict of functions for converting values in certain columns. Keys can either be integers or column labels.

true_valueslist, default None

Values to consider as True.

false_valueslist, default None

Values to consider as False.

skipinitialspaceboolean, default False

Skip spaces after delimiter.

skiprowslist-like or integer, default None

Line numbers to skip (0-indexed) or number of lines to skip (int) at the start of the file.

If callable, the callable function will be evaluated against the row indices, returning True if the row should be skipped and False otherwise:

```
In [10]: data = "col1,col2,col3\na,b,1\na,b,2\nc,d,3"

In [11]: pd.read_csv(StringIO(data))

Out[11]:

  col1 col2  col3

0   a    b    1

1   a    b    2

2   c    d    3


In [12]: pd.read_csv(StringIO(data), skiprows=lambda x: x % 2 != 0)

Out[12]:

  col1 col2  col3

0   a    b    2
```

skipfooterint, default 0

Number of lines at bottom of file to skip (unsupported with engine=’c’).

nrowsint, default None

Number of rows of file to read. Useful for reading pieces of large files.

low_memoryboolean, default True

Internally process the file in chunks, resulting in lower memory use while parsing, but possibly mixed type inference. To ensure no mixed types either set False, or specify the type with the dtype parameter. Note that the entire file is read into a single DataFrame regardless, use the chunksize or iterator parameter to return the data in chunks. (Only valid with C parser)

memory_mapboolean, default False

If a filepath is provided for filepath_or_buffer, map the file object directly onto memory and access the data directly from there. Using this option can improve performance because there is no longer any I/O overhead.

NA and missing data handling
na_valuesscalar, str, list-like, or dict, default None

Additional strings to recognize as NA/NaN. If dict passed, specific per-column NA values. See na values const below for a list of the values interpreted as NaN by default.

keep_default_naboolean, default True

Whether or not to include the default NaN values when parsing the data. Depending on whether na_values is passed in, the behavior is as follows:

- If keep_default_na is True, and na_values are specified, na_values is appended to the default NaN values used for parsing.
- If keep_default_na is True, and na_values are not specified, only the default NaN values are used for parsing.
- If keep_default_na is False, and na_values are specified, only the NaN values specified na_values are used for parsing.
- If keep_default_na is False, and na_values are not specified, no strings will be parsed as NaN.

Note that if na_filter is passed in as False, the keep_default_na and na_values parameters will be ignored.

na_filterboolean, default True

Detect missing value markers (empty strings and the value of na_values). In data without any NAs, passing na_filter=False can improve the performance of reading a large file.

verboseboolean, default False

Indicate number of NA values placed in non-numeric columns.

skip_blank_linesboolean, default True

If True, skip over blank lines rather than interpreting as NaN values.

Datetime handling

parse_datesboolean or list of ints or names or list of lists or dict, default False.

- If True -> try parsing the index.
- If [1, 2, 3] -> try parsing columns 1, 2, 3 each as a separate date column.
- If [[1, 3]] -> combine columns 1 and 3 and parse as a single date column.
- If {'foo': [1, 3]} -> parse columns 1, 3 as date and call result ‘foo’. A fast-path exists for iso8601-formatted dates.

infer_datetime_formatboolean, default False

If True and parse_dates is enabled for a column, attempt to infer the datetime format to speed up the processing.

keep_date_colboolean, default False

If True and parse_dates specifies combining multiple columns then keep the original columns.

date_parserfunction, default None

Function to use for converting a sequence of string columns to an array of datetime instances. The default uses dateutil.parser.parser to do the conversion. pandas will try to call date_parser in three different ways, advancing to the next if an exception occurs: 1) Pass one or more arrays (as defined by parse_dates) as arguments; 2) concatenate (row-wise) the string values from the columns defined by parse_dates into a single array and pass that; and 3) call date_parser once for each row using one or more strings (corresponding to the columns defined by parse_dates) as arguments.

dayfirstboolean, default False

DD/MM format dates, international and European format.

cache_datesboolean, default True

If True, use a cache of unique, converted dates to apply the datetime conversion. May produce significant speed-up when parsing duplicate date strings, especially ones with timezone offsets.

New in version 0.25.0.

Iteration

iteratorboolean, default False

Return TextFileReader object for iteration or getting chunks with get_chunk().

chunksizeint, default None

Return TextFileReader object for iteration. See iterating and chunking below.

Quoting, compression, and file format

compression{'infer', 'gzip', 'bz2', 'zip', 'xz', 'zstd', None, dict}, default 'infer'

For on-the-fly decompression of on-disk data. If ‘infer’, then use gzip, bz2, zip, xz, or zstandard if filepath_or_buffer is path-like ending in ‘.gz’, ‘.bz2’, ‘.zip’, ‘.xz’, ‘.zst’, respectively, and no decompression otherwise. If using ‘zip’, the ZIP file must contain only one data file to be read in. Set to None for no decompression. Can also be a dict with key 'method' set to one of {'zip', 'gzip', 'bz2', 'zstd'} and other key-value pairs are forwarded to zipfile.ZipFile, gzip.GzipFile, bz2.BZ2File, or zstandard.ZstdDecompressor. As an example, the following could be passed for faster compression and to create a reproducible gzip archive: compression={'method': 'gzip', 'compresslevel': 1, 'mtime': 1}.

Changed in version 1.1.0: dict option extended to support gzip and bz2.

Changed in version 1.2.0: Previous versions forwarded dict entries for ‘gzip’ to gzip.open.

thousandsstr, default None

Thousands separator.

decimalstr, default '.'

Character to recognize as decimal point. E.g. use ',' for European data.

float_precisionstring, default None

Specifies which converter the C engine should use for floating-point values. The options are None for the ordinary converter, high for the high-precision converter, and round_trip for the round-trip converter.

lineterminatorstr (length 1), default None

Character to break file into lines. Only valid with C parser.

quotecharstr (length 1)

The character used to denote the start and end of a quoted item. Quoted items can include the delimiter and it will be ignored.

quotingint or csv.QUOTE_* instance, default 0

Control field quoting behavior per csv.QUOTE_* constants. Use one of QUOTE_MINIMAL (0), QUOTE_ALL (1), QUOTE_NONNUMERIC (2) or QUOTE_NONE (3).

doublequoteboolean, default True

When quotechar is specified and quoting is not QUOTE_NONE, indicate whether or not to interpret two consecutive quotechar elements inside a field as a single quotechar element.

escapecharstr (length 1), default None

One-character string used to escape delimiter when quoting is QUOTE_NONE.

commentstr, default None

Indicates remainder of line should not be parsed. If found at the beginning of a line, the line will be ignored altogether. This parameter must be a single character. Like empty lines (as long as skip_blank_lines=True), fully commented lines are ignored by the parameter header but not by skiprows. For example, if comment='#', parsing ‘#empty\na,b,c\n1,2,3’ with header=0 will result in ‘a,b,c’ being treated as the header.

encodingstr, default None

Encoding to use for UTF when reading/writing (e.g. 'utf-8'). List of Python standard encodings.

dialectstr or csv.Dialect instance, default None

If provided, this parameter will override values (default or not) for the following parameters: delimiter, doublequote, escapechar, skipinitialspace, quotechar, and quoting. If it is necessary to override values, a ParserWarning will be issued. See csv.Dialect documentation for more details.

Error handling

error_bad_linesboolean, optional, default None

Lines with too many fields (e.g. a csv line with too many commas) will by default cause an exception to be raised, and no DataFrame will be returned. If False, then these “bad lines” will dropped from the DataFrame that is returned. See bad lines below.

Deprecated since version 1.3.0: The on_bad_lines parameter should be used instead to specify behavior upon encountering a bad line instead.

warn_bad_linesboolean, optional, default None

If error_bad_lines is False, and warn_bad_lines is True, a warning for each “bad line” will be output.

Deprecated since version 1.3.0: The on_bad_lines parameter should be used instead to specify behavior upon encountering a bad line instead.

on_bad_lines(‘error’, ‘warn’, ‘skip’), default ‘error’

Specifies what to do upon encountering a bad line (a line with too many fields). Allowed values are :

- ‘error’, raise an ParserError when a bad line is encountered.

- ‘warn’, print a warning when a bad line is encountered and skip that line.
- ‘skip’, skip bad lines without raising or warning when they are encountered.

New in version 1.3.0.

Specifying column data types

You can indicate the data type for the whole DataFrame or individual columns:

```
In [13]: import numpy as np

In [14]: data = "a,b,c,d\n1,2,3,4\n5,6,7,8\n9,10,11"

In [15]: print(data)

a,b,c,d

1,2,3,4

5,6,7,8

9,10,11


In [16]: df = pd.read_csv(StringIO(data), dtype=object)

In [17]: df

Out[17]:

   a  b   c   d

0  1  2   3   4

1  5  6   7   8

2  9  10  11  NaN
```

```
In [18]: df["a"][0]

Out[18]: '1'

In [19]: df = pd.read_csv(StringIO(data), dtype={"b": object, "c": np.float64, "d": "Int64"})

In [20]: df.dtypes

Out[20]:

a      int64

b     object

c    float64

d      Int64
```

dtype: object

Fortunately, pandas offers more than one way to ensure that your column(s) contain only one dtype. If you're unfamiliar with these concepts, you can see here to learn more about dtypes, and here to learn more about object conversion in pandas.

For instance, you can use the converters argument of read_csv():

```
In [21]: data = "col_1\n1\n2\n'A'\n4.22"


In [22]: df = pd.read_csv(StringIO(data), converters={"col_1": str})

In [23]: df

Out[23]:
  col_1
0    1
1    2
2  'A'
3  4.22
In [24]: df["col_1"].apply(type).value_counts()
Out[24]:
<class 'str'>    4
Name: col_1, dtype: int64
```

Or you can use the to_numeric() function to coerce the dtypes after reading in the data,

```
In [25]: df2 = pd.read_csv(StringIO(data))
In [26]: df2["col_1"] = pd.to_numeric(df2["col_1"], errors="coerce")


In [27]: df2
Out[27]:
   col_1
0   1.00
1   2.00
2    NaN
3   4.22
```

```
In [28]: df2["col_1"].apply(type).value_counts()

Out[28]:

<class 'float'>    4

Name: col_1, dtype: int64
```

which will convert all valid parsing to floats, leaving the invalid parsing as NaN.

Ultimately, how you deal with reading in columns containing mixed dtypes depends on your specific needs. In the case above, if you wanted to NaN out the data anomalies, then to_numeric() is probably your best option. However, if you wanted for all the data to be coerced, no matter the type, then using the converters argument of read_csv() would certainly be worth trying.

Note

In some cases, reading in abnormal data with columns containing mixed dtypes will result in an inconsistent dataset. If you rely on pandas to infer the dtypes of your columns, the parsing engine will go and infer the dtypes for different chunks of the data, rather than the whole dataset at once. Consequently, you can end up with column(s) with mixed dtypes. For example,

```
In [29]: col_1 = list(range(500000)) + ["a", "b"] + list(range(500000))


In [30]: df = pd.DataFrame({"col_1": col_1})

In [31]: df.to_csv("foo.csv")

In [32]: mixed_df = pd.read_csv("foo.csv")
In [33]: mixed_df["col_1"].apply(type).value_counts()

Out[33]:

<class 'int'>    737858

<class 'str'>    262144

Name: col_1, dtype: int64


In [34]: mixed_df["col_1"].dtype

Out[34]: dtype('O')
```

will result with mixed_df containing an int dtype for certain chunks of the column, and str for others due to the mixed dtypes from the data that was read in. It is important to note that the overall column will be marked with a dtype of object, which is used for columns with mixed dtypes.

Specifying categorical dtype

Categorical columns can be parsed directly by specifying dtype='category' or dtype=CategoricalDtype(categories, ordered).

```
In [35]: data = "col1,col2,col3\na,b,1\na,b,2\nc,d,3"


In [36]: pd.read_csv(StringIO(data))

Out[36]:

 col1 col2  col3

0    a    b    1

1    a    b    2
```

```
2    c    d    3

In [37]: pd.read_csv(StringIO(data)).dtypes
Out[37]:
col1    object
col2    object
col3     int64
dtype: object

In [38]: pd.read_csv(StringIO(data), dtype="category").dtypes
Out[38]:
col1    category
col2    category
col3    category
dtype: object
```

Individual columns can be parsed as a Categorical using a dict specification:

```
In [39]: pd.read_csv(StringIO(data), dtype={"col1": "category"}).dtypes
Out[39]:
col1    category
col2      object
col3       int64
```

```
dtype: object
```

Specifying dtype='category' will result in an unordered Categorical whose categories are the unique values observed in the data. For more control on the categories and order, create a CategoricalDtype ahead of time, and pass that for that column’s dtype.

```
In [40]: from pandas.api.types import CategoricalDtype


In [41]: dtype = CategoricalDtype(["d", "c", "b", "a"], ordered=True)

In [42]: pd.read_csv(StringIO(data), dtype={"col1": dtype}).dtypes

Out[42]:

col1    category

col2      object

col3       int64

dtype: object
```

When using dtype=CategoricalDtype, “unexpected” values outside of dtype.categories are treated as missing values.

```
In [43]: dtype = CategoricalDtype(["a", "b", "d"])  # No 'c'

In [44]: pd.read_csv(StringIO(data), dtype={"col1": dtype}).col1

Out[44]:

0      a

1      a
```

```
2    NaN

Name: col1, dtype: category

Categories (3, object): ['a', 'b', 'd']
```

This matches the behavior of Categorical.set_categories().

Note

With dtype='category', the resulting categories will always be parsed as strings (object dtype). If the categories are numeric they can be converted using the to_numeric() function, or as appropriate, another converter such as to_datetime().

When dtype is a CategoricalDtype with homogeneous categories (all numeric, all datetimes, etc.), the conversion is done automatically.

```
In [45]: df = pd.read_csv(StringIO(data), dtype="category")

In [46]: df.dtypes

Out[46]:

col1    category

col2    category

col3    category

dtype: object
```

```
In [47]: df["col3"]

Out[47]:

0    1

1    2

2    3

Name: col3, dtype: category

Categories (3, object): ['1', '2', '3']


In [48]: df["col3"].cat.categories = pd.to_numeric(df["col3"].cat.categories)


In [49]: df["col3"]

Out[49]:

0    1

1    2

2    3

Name: col3, dtype: category

Categories (3, int64): [1, 2, 3]
```

Naming and using columns

Handling column names

A file may or may not have a header row. pandas assumes the first row should be used as the column names:

```
In [50]: data = "a,b,c\n1,2,3\n4,5,6\n7,8,9"

In [51]: print(data)

a,b,c

1,2,3

4,5,6

7,8,9


In [52]: pd.read_csv(StringIO(data))

Out[52]:

   a  b  c

0  1  2  3

1  4  5  6

2  7  8  9
```

By specifying the names argument in conjunction with header you can indicate other names to use and whether or not to throw away the header row (if any):

```
In [53]: print(data)

a,b,c
```

```
1,2,3

4,5,6

7,8,9


In [54]: pd.read_csv(StringIO(data), names=["foo", "bar", "baz"], header=0)

Out[54]:

   foo  bar  baz

0    1    2    3

1    4    5    6

2    7    8    9


In [55]: pd.read_csv(StringIO(data), names=["foo", "bar", "baz"],
header=None)

Out[55]:

  foo bar baz

0   a   b   c

1   1   2   3

2   4   5   6

3   7   8   9
```

If the header is in a row other than the first, pass the row number to header. This will skip the preceding rows:

```
In [56]: data = "skip this skip it\na,b,c\n1,2,3\n4,5,6\n7,8,9"
In [57]: pd.read_csv(StringIO(data), header=1)
```

```
Out[57]:

  a  b  c

0  1  2  3

1  4  5  6

2  7  8  9
```

Note

Default behavior is to infer the column names: if no names are passed the behavior is identical to header=0 and column names are inferred from the first non-blank line of the file, if column names are passed explicitly then the behavior is identical to header=None.

Duplicate names parsing

If the file or header contains duplicate names, pandas will by default distinguish between them so as to prevent overwriting data:

```
In [58]: data = "a,b,a\n0,1,2\n3,4,5"
```

```
In [59]: pd.read_csv(StringIO(data))
```

```
Out[59]:

  a  b  a.1

0  0  1    2

1  3  4    5
```

There is no more duplicate data because mangle_dupe_cols=True by default, which modifies a series of duplicate columns ‘X’, …, ‘X’ to become ‘X’, ‘X.1’, …, ‘X.N’. If mangle_dupe_cols=False, duplicate data can arise:

```
In [2]: data = 'a,b,a\n0,1,2\n3,4,5'

In [3]: pd.read_csv(StringIO(data), mangle_dupe_cols=False)

Out[3]:

   a  b  a

0  2  1  2

1  5  4  5
```

To prevent users from encountering this problem with duplicate data, a ValueError exception is raised if mangle_dupe_cols != True:

```
In [2]: data = 'a,b,a\n0,1,2\n3,4,5'

In [3]: pd.read_csv(StringIO(data), mangle_dupe_cols=False)

...

ValueError: Setting mangle_dupe_cols=False is not supported yet
```

Filtering columns (usecols)

The usecols argument allows you to select any subset of the columns in a file, either using the column names, position numbers or a callable:

```
In [60]: data = "a,b,c,d\n1,2,3,foo\n4,5,6,bar\n7,8,9,baz"
In [61]: pd.read_csv(StringIO(data))
Out[61]:
   a  b  c    d
0  1  2  3  foo
1  4  5  6  bar
2  7  8  9  baz

In [62]: pd.read_csv(StringIO(data), usecols=["b", "d"])
Out[62]:
   b    d
0  2  foo
1  5  bar
2  8  baz

In [63]: pd.read_csv(StringIO(data), usecols=[0, 2, 3])
Out[63]:
   a  c    d
0  1  3  foo
1  4  6  bar
2  7  9  baz
```

```
In [64]: pd.read_csv(StringIO(data), usecols=lambda x: x.upper() in ["A",
"C"])

Out[64]:

   a  c

0  1  3

1  4  6

2  7  9
```

The usecols argument can also be used to specify which columns not to use in the final result:

```
In [65]: pd.read_csv(StringIO(data), usecols=lambda x: x not in ["a", "c"])

Out[65]:

   b    d

0  2  foo

1  5  bar

2  8  baz
```

In this case, the callable is specifying that we exclude the “a” and “c” columns from the output.

Comments and empty lines

Ignoring line comments and empty lines
If the comment parameter is specified, then completely commented lines will be ignored. By default, completely blank lines will be ignored as well.

```
In [66]: data = "\na,b,c\n  \n# commented line\n1,2,3\n\n4,5,6"
In [67]: print(data)

a,b,c

# commented line
1,2,3
4,5,6
In [68]: pd.read_csv(StringIO(data), comment="#")
Out[68]:
   a  b  c
0  1  2  3
1  4  5  6
```

If skip_blank_lines=False, then read_csv will not ignore blank lines:

```
In [69]: data = "a,b,c\n\n1,2,3\n\n\n4,5,6"
In [70]: pd.read_csv(StringIO(data), skip_blank_lines=False)
Out[70]:
     a    b    c
0  NaN  NaN  NaN
1  1.0  2.0  3.0
2  NaN  NaN  NaN
3  NaN  NaN  NaN
```

```
4  4.0  5.0  6.0
```

Warning

The presence of ignored lines might create ambiguities involving line numbers; the parameter header uses row numbers (ignoring commented/empty lines), while skiprows uses line numbers (including commented/empty lines):

```
In [71]: data = "#comment\na,b,c\nA,B,C\n1,2,3"

In [72]: pd.read_csv(StringIO(data), comment="#", header=1)

Out[72]:

   A  B  C

0  1  2  3

In [73]: data = "A,B,C\n#comment\na,b,c\n1,2,3"

In [74]: pd.read_csv(StringIO(data), comment="#", skiprows=2)

Out[74]:

   a  b  c

0  1  2  3
```

If both header and skiprows are specified, header will be relative to the end of skiprows. For example:

```
In [75]: data = (

   ....:     "# empty\n"
   ....:     "# second empty line\n"
   ....:     "# third emptyline\n"
   ....:     "X,Y,Z\n"
   ....:     "1,2,3\n"
   ....:     "A,B,C\n"
   ....:     "1,2.,4.\n"
   ....:     "5.,NaN,10.0\n"
   ....: )
   ....:

In [76]: print(data)
# empty
# second empty line
# third emptyline
X,Y,Z
1,2,3
A,B,C
1,2.,4.
5.,NaN,10.0


In [77]: pd.read_csv(StringIO(data), comment="#", skiprows=4, header=1)
```

```
Out[77]:
     A    B     C
0  1.0  2.0   4.0
1  5.0  NaN  10.0
```

Comments

Sometimes comments or meta data may be included in a file:

```
In [78]: print(open("tmp.csv").read())
ID,level,category
Patient1,123000,x # really unpleasant
Patient2,23000,y # wouldn't take his medicine
Patient3,1234018,z # awesome
```

By default, the parser includes the comments in the output:

```
In [79]: df = pd.read_csv("tmp.csv")

In [80]: df
Out[80]:
         ID    level                        category
0  Patient1   123000           x # really unpleasant
1  Patient2    23000  y # wouldn't take his medicine
2  Patient3  1234018                     z # awesome
```

We can suppress the comments using the comment keyword:

```
In [81]: df = pd.read_csv("tmp.csv", comment="#")

In [82]: df

Out[82]:

        ID    level category

0  Patient1   123000        x

1  Patient2    23000        y

2  Patient3  1234018        z
```

Dealing with Unicode data

The encoding argument should be used for encoded unicode data, which will result in byte strings being decoded to unicode in the result:

```
In [83]: from io import BytesIO

In [84]: data = b"word,length\n" b"Tr\xc3\xa4umen,7\n"
b"Gr\xc3\xbc\xc3\x9fe,5"

In [85]: data = data.decode("utf8").encode("latin-1")

In [86]: df = pd.read_csv(BytesIO(data), encoding="latin-1")

In [87]: df

Out[87]:

     word  length

0  Träumen       7
```

```
1   Grüße      5
```

```
In [88]: df["word"][1]

Out[88]: 'Grüße'
```

Some formats which encode all characters as multiple bytes, like UTF-16, won’t parse correctly at all without specifying the encoding. Full list of Python standard encodings.

Index columns and trailing delimiters

If a file has one more column of data than the number of column names, the first column will be used as the DataFrame’s row names:

```
In [89]: data = "a,b,c\n4,apple,bat,5.7\n8,orange,cow,10"

In [90]: pd.read_csv(StringIO(data))

Out[90]:

       a    b     c

4   apple  bat   5.7

8  orange  cow  10.0
```

```
In [91]: data = "index,a,b,c\n4,apple,bat,5.7\n8,orange,cow,10"

In [92]: pd.read_csv(StringIO(data), index_col=0)

Out[92]:

           a    b     c

index
```

```
4      apple  bat   5.7

8     orange  cow  10.0
```

Ordinarily, you can achieve this behavior using the index_col option.

There are some exception cases when a file has been prepared with delimiters at the end of each data line, confusing the parser. To explicitly disable the index column inference and discard the last column, pass index_col=False:

```
In [93]: data = "a,b,c\n4,apple,bat,\n8,orange,cow,"

In [94]: print(data)

a,b,c

4,apple,bat,

8,orange,cow,

In [95]: pd.read_csv(StringIO(data))

Out[95]:

        a    b   c

4   apple  bat NaN

8  orange  cow NaN



In [96]: pd.read_csv(StringIO(data), index_col=False)

Out[96]:

   a      b    c
```

```
0  4   apple  bat
1  8  orange  cow
```

If a subset of data is being parsed using the usecols option, the index_col specification is based on that subset, not the original data.

```
In [97]: data = "a,b,c\n4,apple,bat,\n8,orange,cow,"

In [98]: print(data)
a,b,c
4,apple,bat,
8,orange,cow,


In [99]: pd.read_csv(StringIO(data), usecols=["b", "c"])
Out[99]:
     b   c
4  bat NaN
8  cow NaN


In [100]: pd.read_csv(StringIO(data), usecols=["b", "c"], index_col=0)
Out[100]:
     b   c
4  bat NaN
8  cow NaN
```

Date Handling

Specifying date columns

To better facilitate working with datetime data, read_csv() uses the keyword arguments parse_dates and date_parser to allow users to specify a variety of columns and date/time formats to turn the input text data into datetime objects.

The simplest case is to just pass in parse_dates=True:

```
# Use a column as an index, and parse it as dates.

In [101]: df = pd.read_csv("foo.csv", index_col=0, parse_dates=True)

In [102]: df

Out[102]:

            A  B  C

date

2009-01-01  a  1  2

2009-01-02  b  3  4

2009-01-03  c  4  5


# These are Python datetime objects

In [103]: df.index

Out[103]: DatetimeIndex(['2009-01-01', '2009-01-02', '2009-01-03'],
dtype='datetime64[ns]', name='date', freq=None)
```

It is often the case that we may want to store date and time data separately, or store various date fields separately. the parse_dates keyword can be used to specify a combination of columns to parse the dates and/or times from.

You can specify a list of column lists to parse_dates, the resulting date columns will be prepended to the output (so as to not affect the existing column order) and the new column names will be the concatenation of the component column names:

```
In [104]: print(open("tmp.csv").read())

KORD,19990127, 19:00:00, 18:56:00, 0.8100

KORD,19990127, 20:00:00, 19:56:00, 0.0100

KORD,19990127, 21:00:00, 20:56:00, -0.5900

KORD,19990127, 21:00:00, 21:18:00, -0.9900

KORD,19990127, 22:00:00, 21:56:00, -0.5900

KORD,19990127, 23:00:00, 22:56:00, -0.5900


In [105]: df = pd.read_csv("tmp.csv", header=None, parse_dates=[[1, 2], [1,
3]])

In [106]: df

Out[106]:

                  1_2                 1_3     0     4

0 1999-01-27 19:00:00 1999-01-27 18:56:00  KORD  0.81

1 1999-01-27 20:00:00 1999-01-27 19:56:00  KORD  0.01

2 1999-01-27 21:00:00 1999-01-27 20:56:00  KORD -0.59

3 1999-01-27 21:00:00 1999-01-27 21:18:00  KORD -0.99
```

```
4 1999-01-27 22:00:00 1999-01-27 21:56:00  KORD -0.59

5 1999-01-27 23:00:00 1999-01-27 22:56:00  KORD -0.59
```

By default the parser removes the component date columns, but you can choose to retain them via the keep_date_col keyword:

```
In [107]: df = pd.read_csv(

   .....:     "tmp.csv", header=None, parse_dates=[[1, 2], [1, 3]], keep_date_col=True

   .....: )

   .....:


In [108]: df

Out[108]:

                  1_2                  1_3     0         1          2          3     4

0 1999-01-27 19:00:00 1999-01-27 18:56:00  KORD  19990127   19:00:00
18:56:00  0.81

1 1999-01-27 20:00:00 1999-01-27 19:56:00  KORD  19990127   20:00:00
19:56:00  0.01

2 1999-01-27 21:00:00 1999-01-27 20:56:00  KORD  19990127   21:00:00
20:56:00 -0.59

3 1999-01-27 21:00:00 1999-01-27 21:18:00  KORD  19990127   21:00:00
21:18:00 -0.99

4 1999-01-27 22:00:00 1999-01-27 21:56:00  KORD  19990127   22:00:00
21:56:00 -0.59
```

```
5 1999-01-27 23:00:00 1999-01-27 22:56:00  KORD  19990127   23:00:00
22:56:00 -0.59
```

Note that if you wish to combine multiple columns into a single date column, a nested list must be used. In other words, parse_dates=[1, 2] indicates that the second and third columns should each be parsed as separate date columns while parse_dates=[[1, 2]] means the two columns should be parsed into a single column.

You can also use a dict to specify custom name columns:

```
In [109]: date_spec = {"nominal": [1, 2], "actual": [1, 3]}

In [110]: df = pd.read_csv("tmp.csv", header=None, parse_dates=date_spec)

In [111]: df

Out[111]:

              nominal              actual     0     4

0 1999-01-27 19:00:00 1999-01-27 18:56:00  KORD  0.81

1 1999-01-27 20:00:00 1999-01-27 19:56:00  KORD  0.01

2 1999-01-27 21:00:00 1999-01-27 20:56:00  KORD -0.59

3 1999-01-27 21:00:00 1999-01-27 21:18:00  KORD -0.99

4 1999-01-27 22:00:00 1999-01-27 21:56:00  KORD -0.59

5 1999-01-27 23:00:00 1999-01-27 22:56:00  KORD -0.59
```

It is important to remember that if multiple text columns are to be parsed into a single date column, then a new column is prepended to the data. The index_col specification is based off of this new set of columns rather than the original data columns:

```
In [112]: date_spec = {"nominal": [1, 2], "actual": [1, 3]}

In [113]: df = pd.read_csv(
   .....:     "tmp.csv", header=None, parse_dates=date_spec, index_col=0
   .....: )  # index is the nominal column
   .....: 

In [114]: df
Out[114]: 
                                 actual     0     4
nominal                                            
1999-01-27 19:00:00 1999-01-27 18:56:00  KORD  0.81
1999-01-27 20:00:00 1999-01-27 19:56:00  KORD  0.01
1999-01-27 21:00:00 1999-01-27 20:56:00  KORD -0.59
1999-01-27 21:00:00 1999-01-27 21:18:00  KORD -0.99
1999-01-27 22:00:00 1999-01-27 21:56:00  KORD -0.59
1999-01-27 23:00:00 1999-01-27 22:56:00  KORD -0.59
```

Note

If a column or index contains an unparsable date, the entire column or index will be returned unaltered as an object data type. For non-standard datetime parsing, use to_datetime() after pd.read_csv.

Note

read_csv has a fast_path for parsing datetime strings in iso8601 format, e.g "2000-01-01T00:01:02+00:00" and similar variations. If you can arrange for your data to store datetimes in this format, load times will be significantly faster, ~20x has been observed.

Date parsing functions

Finally, the parser allows you to specify a custom date_parser function to take full advantage of the flexibility of the date parsing API:

```
In [115]: df = pd.read_csv(
   .....:     "tmp.csv", header=None, parse_dates=date_spec, date_-
parser=pd.to_datetime
   .....: )
   .....:


In [116]: df
Out[116]:
             nominal              actual     0     4
0 1999-01-27 19:00:00 1999-01-27 18:56:00  KORD  0.81
1 1999-01-27 20:00:00 1999-01-27 19:56:00  KORD  0.01
2 1999-01-27 21:00:00 1999-01-27 20:56:00  KORD -0.59
3 1999-01-27 21:00:00 1999-01-27 21:18:00  KORD -0.99
4 1999-01-27 22:00:00 1999-01-27 21:56:00  KORD -0.59
5 1999-01-27 23:00:00 1999-01-27 22:56:00  KORD -0.59
```

pandas will try to call the date_parser function in three different ways. If an exception is raised, the next one is tried:

1. date_parser is first called with one or more arrays as arguments, as defined using parse_dates (e.g., date_parser(['2013', '2013'], ['1', '2'])).
2. If #1 fails, date_parser is called with all the columns concatenated row-wise into a single array (e.g., date_parser(['2013 1', '2013 2'])).

Note that performance-wise, you should try these methods of parsing dates in order:

1. Try to infer the format using infer_datetime_format=True (see section below).
2. If you know the format, use pd.to_datetime(): date_parser=lambda x: pd.to_datetime(x, format=...).
3. If you have a really non-standard format, use a custom date_parser function. For optimal performance, this should be vectorized, i.e., it should accept arrays as arguments.

Parsing a CSV with mixed timezones

pandas cannot natively represent a column or index with mixed timezones. If your CSV file contains columns with a mixture of timezones, the default result will be an object-dtype column with strings, even with parse_dates.

```
In [117]: content = """\

  .....: a

  .....: 2000-01-01T00:00:00+05:00

  .....: 2000-01-01T00:00:00+06:00"""

  .....:


In [118]: df = pd.read_csv(StringIO(content), parse_dates=["a"])
In [119]: df["a"]

Out[119]:

0    2000-01-01 00:00:00+05:00

1    2000-01-01 00:00:00+06:00

Name: a, dtype: object
```

To parse the mixed-timezone values as a datetime column, pass a partially-applied to_datetime() with utc=True as the date_parser.

```
In [120]: df = pd.read_csv(

   .....:     StringIO(content),

   .....:     parse_dates=["a"],

   .....:     date_parser=lambda col: pd.to_datetime(col, utc=True),

   .....: )

   .....:

In [121]: df["a"]

Out[121]:

0   1999-12-31 19:00:00+00:00

1   1999-12-31 18:00:00+00:00

Name: a, dtype: datetime64[ns, UTC]
```

Inferring datetime format

If you have parse_dates enabled for some or all of your columns, and your datetime strings are all formatted the same way, you may get a large speed up by setting infer_datetime_format=True. If set, pandas will attempt to guess the format of your datetime strings, and then use a faster means of parsing the strings. 5-10x parsing speeds have been observed. pandas will fallback to the usual parsing if either the format cannot be guessed or the format that was guessed cannot properly parse the entire column of strings. So in general, infer_datetime_format should not have any negative consequences if enabled.

Here are some examples of datetime strings that can be guessed (All representing December 30th, 2011 at 00:00:00):

- “20111230”
- “2011/12/30”
- “20111230 00:00:00”
- “12/30/2011 00:00:00”
- “30/Dec/2011 00:00:00”
- “30/December/2011 00:00:00”

Note that infer_datetime_format is sensitive to dayfirst. With dayfirst=True, it will guess “01/12/2011” to be December 1st. With dayfirst=False (default) it will guess “01/12/2011” to be January 12th.

```
# Try to infer the format for the index column

In [122]: df = pd.read_csv(

   .....:    "foo.csv",

   .....:    index_col=0,

   .....:    parse_dates=True,

   .....:    infer_datetime_format=True,

   .....: )

   .....:
```

```
In [123]: df

Out[123]:

            A  B  C

date

2009-01-01  a  1  2

2009-01-02  b  3  4

2009-01-03  c  4  5
```

International date formats

While US date formats tend to be MM/DD/YYYY, many international formats use DD/MM/YYYY instead. For convenience, a dayfirst keyword is provided:

```
In [124]: print(open("tmp.csv").read())

date,value,cat

1/6/2000,5,a

2/6/2000,10,b

3/6/2000,15,c



In [125]: pd.read_csv("tmp.csv", parse_dates=[0])

Out[125]:

        date  value cat

0 2000-01-06      5   a

1 2000-02-06     10   b
```

```
2 2000-03-06     15   c
```

```
In [126]: pd.read_csv("tmp.csv", dayfirst=True, parse_dates=[0])
Out[126]:
        date  value cat
0 2000-06-01      5   a
1 2000-06-02     10   b
2 2000-06-03     15   c
```

Writing CSVs to binary file objects

New in version 1.2.0.

df.to_csv(..., mode="wb") allows writing a CSV to a file object opened binary mode. In most cases, it is not necessary to specify mode as Pandas will auto-detect whether the file object is opened in text or binary mode.

```
In [127]: import io

In [128]: data = pd.DataFrame([0, 1, 2])

In [129]: buffer = io.BytesIO()

In [130]: data.to_csv(buffer, encoding="utf-8", compression="gzip")
```

Specifying method for floating-point conversion

The parameter float_precision can be specified in order to use a specific floating-point converter during parsing with the C engine. The options are the ordinary converter, the high-precision converter, and the round-trip converter (which is guaranteed to round-trip values after writing to a file). For example:

```
In [131]: val =
"0.3066101993807095471566981359501369297504425048828125"

In [132]: data = "a,b,c\n1,2,{0}".format(val)

In [133]: abs(

   ......:     pd.read_csv(

   ......:         StringIO(data),

   ......:         engine="c",

   ......:         float_precision=None,

   ......:     )["c"][0] - float(val)

   ......: )

   ......:

Out[133]: 5.551115123125783e-17


In [134]: abs(

   ......:     pd.read_csv(

   ......:         StringIO(data),
   .....:         engine="c",
   .....:         float_precision="high",
   .....:     )["c"][0] - float(val)
   .....: )
   .....:
Out[134]: 5.551115123125783e-17

In [135]: abs(
   .....:     pd.read_csv(StringIO(data), engine="c", float_precision="round_trip")["c"][0]
   .....:     - float(val)
   .....: )
   .....:
Out[135]: 0.0
```

Thousand separators

For large numbers that have been written with a thousands separator, you can set the thousands keyword to a string of length 1 so that integers will be parsed correctly:

By default, numbers with a thousands separator will be parsed as strings:

```
In [136]: print(open("tmp.csv").read())
ID|level|category
```

```
Patient1|123,000|x

Patient2|23,000|y

Patient3|1,234,018|z

In [137]: df = pd.read_csv("tmp.csv", sep="|")

In [138]: df

Out[138]:

        ID      level category

0  Patient1    123,000        x

1  Patient2     23,000        y

2  Patient3  1,234,018        z


In [139]: df.level.dtype

Out[139]: dtype('O')
```

The thousands keyword allows integers to be parsed correctly:

```
In [140]: print(open("tmp.csv").read())

ID|level|category

Patient1|123,000|x

Patient2|23,000|y

Patient3|1,234,018|z


In [141]: df = pd.read_csv("tmp.csv", sep="|", thousands=",")
In [142]: df

Out[142]:

        ID    level category

0  Patient1   123000        x

1  Patient2    23000        y

2  Patient3  1234018        z


In [143]: df.level.dtype

Out[143]: dtype('int64')
```

NA values

To control which values are parsed as missing values (which are signified by NaN), specify a string in na_values. If you specify a list of strings, then all values in it are considered to be missing values. If you specify a number (a float, like 5.0 or an integer like 5), the corresponding equivalent values will also imply a missing value (in this case effectively [5.0, 5] are recognized as NaN).

To completely override the default values that are recognized as missing, specify keep_default_na=False.

The default NaN recognized values are ['-1.#IND', '1.#QNAN', '1.#IND', '-1.#QNAN', '#N/A N/A', '#N/A', 'N/A', 'n/a', 'NA', '<NA>', '#NA', 'NULL', 'null', 'NaN', '-NaN', 'nan', '-nan', ''].

Let us consider some examples:

```
pd.read_csv("path_to_file.csv", na_values=[5])
```

In the example above 5 and 5.0 will be recognized as NaN, in addition to the defaults. A string will first be interpreted as a numerical 5, then as a NaN.

```
pd.read_csv("path_to_file.csv", keep_default_na=False, na_values=[""])
```

Above, only an empty field will be recognized as NaN.

```
pd.read_csv("path_to_file.csv", keep_default_na=False, na_values=["NA",
"0"])
```

Above, both NA and 0 as strings are NaN.

```
pd.read_csv("path_to_file.csv", na_values=["Nope"])
```

The default values, in addition to the string "Nope" are recognized as NaN.

Infinity

inf like values will be parsed as np.inf (positive infinity), and -inf as -np.inf (negative infinity). These will ignore the case of the value, meaning Inf, will also be parsed as np.inf.

Returning Series

Using the squeeze keyword, the parser will return output with a single column as a Series:

Deprecated since version 1.4.0: Users should append .squeeze("columns") to the DataFrame returned by read_csv instead.

```
In [144]: print(open("tmp.csv").read())
```

```
level

Patient1,123000
```

```
Patient2,23000

Patient3,1234018


In [145]: output = pd.read_csv("tmp.csv", squeeze=True)

In [146]: output

Out[146]:

Patient1     123000

Patient2      23000

Patient3    1234018

Name: level, dtype: int64


In [147]: type(output)

Out[147]: pandas.core.series.Series
```

Boolean values

The common values True, False, TRUE, and FALSE are all recognized as boolean. Occasionally you might want to recognize other values as being boolean. To do this, use the true_values and false_values options as follows:

```
In [148]: data = "a,b,c\n1,Yes,2\n3,No,4"

In [149]: print(data)

a,b,c
```

```
1,Yes,2

3,No,4


In [150]: pd.read_csv(StringIO(data))

Out[150]:

   a    b  c

0  1  Yes  2

1  3   No  4


In [151]: pd.read_csv(StringIO(data), true_values=["Yes"], false_values=
["No"])

Out[151]:

   a      b  c

0  1   True  2

1  3  False  4
```

Handling “bad” lines

Some files may have malformed lines with too few fields or too many. Lines with too few fields will have NA values filled in the trailing fields. Lines with too many fields will raise an error by default:

```
In [152]: data = "a,b,c\n1,2,3\n4,5,6,7\n8,9,10"

In [153]: pd.read_csv(StringIO(data))

---------------------------------------------------------------------------
```

```
ParserError                               Traceback (most recent call last)
Input In [153], in <module>
----> 1 pd.read_csv(StringIO(data))

File /pandas/pandas/util/_decorators.py:311, in deprecate_nonkeyword_arguments.<locals>.decorate.<locals>.wrapper(*args, **kwargs)
    305 if len(args) > num_allow_args:
    306     warnings.warn(
    307         msg.format(arguments=arguments),
    308         FutureWarning,
    309         stacklevel=stacklevel,
    310     )
--> 311 return func(*args, **kwargs)

File /pandas/pandas/io/parsers/readers.py:680, in read_csv(filepath_or_buffer, sep, delimiter, header, names, index_col, usecols, squeeze, prefix, mangle_dupe_cols, dtype, engine, converters, true_values, false_values, skipinitialspace, skiprows, skipfooter, nrows, na_values, keep_default_na, na_filter, verbose, skip_blank_lines, parse_dates, infer_datetime_format, keep_date_col, date_parser, dayfirst, cache_dates, iterator, chunksize, compression, thousands, decimal, lineterminator, quotechar, quoting, doublequote, escapechar, comment, encoding, encoding_errors, dialect, error_bad_lines, warn_bad_lines, on_bad_lines, delim_whitespace, low_memory, memory_map, float_precision, storage_options)
    665 kwds_defaults = _refine_defaults_read(
    666     dialect,
    667     delimiter,
   (...)
    676     defaults={"delimiter": ","},
    677 )
    678 kwds.update(kwds_defaults)
--> 680 return _read(filepath_or_buffer, kwds)


File /pandas/pandas/io/parsers/readers.py:581, in _read(filepath_or_buffer,
kwds)
    578     return parser
    580 with parser:
--> 581     return parser.read(nrows)


File /pandas/pandas/io/parsers/readers.py:1250, in TextFileReader.read(self,
nrows)
   1248 nrows = validate_integer("nrows", nrows)
   1249 try:
-> 1250     index, columns, col_dict = self._engine.read(nrows)
   1251 except Exception:
   1252     self.close()
```

```
File /pandas/pandas/io/parsers/c_parser_wrapper.py:225, in CParserWrapper.read(self, nrows)
    223 try:
    224     if self.low_memory:
--> 225         chunks = self._reader.read_low_memory(nrows)
    226         # destructive to chunks
    227         data = _concatenate_chunks(chunks)

File /pandas/pandas/_libs/parsers.pyx:805, in pandas._libs.parsers.TextReader.read_low_memory()

File /pandas/pandas/_libs/parsers.pyx:861, in pandas._libs.parsers.TextReader._read_rows()

File /pandas/pandas/_libs/parsers.pyx:847, in pandas._libs.parsers.TextReader._tokenize_rows()

File /pandas/pandas/_libs/parsers.pyx:1960, in pandas._libs.parsers.raise_parser_error()

ParserError: Error tokenizing data. C error: Expected 3 fields in line 3, saw 4
```

You can elect to skip bad lines:

```
In [29]: pd.read_csv(StringIO(data), on_bad_lines="warn")
Skipping line 3: expected 3 fields, saw 4

Out[29]: 
   a  b   c
0  1  2   3
1  8  9  10
```

Or pass a callable function to handle the bad line if engine="python". The bad line will be a list of strings that was split by the sep:

```
In [29]: external_list = []


In [30]: def bad_lines_func(line):

   ...:     external_list.append(line)

   ...:     return line[-3:]


In [31]: pd.read_csv(StringIO(data), on_bad_lines=bad_lines_func, engine="python")

Out[31]:

   a  b   c

0  1  2   3

1  5  6   7

2  8  9  10


In [32]: external_list

Out[32]: [4, 5, 6, 7]
```

You can also use the usecols parameter to eliminate extraneous column data that appear in some lines but not others:

```
In [33]: pd.read_csv(StringIO(data), usecols=[0, 1, 2])

Out[33]:

   a  b   c

0  1  2   3

1  4  5   6

2  8  9  10
```

In case you want to keep all data including the lines with too many fields, you can specify a sufficient number of names. This ensures that lines with not enough fields are filled with NaN.

```
In [34]: pd.read_csv(StringIO(data), names=['a', 'b', 'c', 'd'])

Out[34]:

   a  b   c  d

0  1  2   3  NaN

1  4  5   6  7

2  8  9  10  NaN
```

Dialect

The dialect keyword gives greater flexibility in specifying the file format. By default it uses the Excel dialect but you can specify either the dialect name or a csv.Dialect instance.

Suppose you had data with unenclosed quotes:

```
In [154]: print(data)
```

```
label1,label2,label3

index1,"a,c,e

index2,b,d,f
```

By default, read_csv uses the Excel dialect and treats the double quote as the quote character, which causes it to fail when it finds a newline before it finds the closing double quote.

We can get around this using dialect:

```
In [155]: import csv

In [156]: dia = csv.excel()

In [157]: dia.quoting = csv.QUOTE_NONE

In [158]: pd.read_csv(StringIO(data), dialect=dia)

Out[158]:

       label1 label2 label3

index1     "a      c      e

index2      b      d      f
```

All of the dialect options can be specified separately by keyword arguments:

```
In [159]: data = "a,b,c~1,2,3~4,5,6"


In [160]: pd.read_csv(StringIO(data), lineterminator="~")

Out[160]:

  a  b  c
```

```
0  1  2  3

1  4  5  6
```

Another common dialect option is skipinitialspace, to skip any whitespace after a delimiter:

```
In [161]: data = "a, b, c\n1, 2, 3\n4, 5, 6"

In [162]: print(data)

a, b, c

1, 2, 3

4, 5, 6

In [163]: pd.read_csv(StringIO(data), skipinitialspace=True)

Out[163]:

   a  b  c

0  1  2  3

1  4  5  6
```

The parsers make every attempt to “do the right thing” and not be fragile. Type inference is a pretty big deal. If a column can be coerced to integer dtype without altering the contents, the parser will do so. Any non-numeric columns will come through as object dtype as with the rest of pandas objects.

Quoting and Escape Characters

Quotes (and other escape characters) in embedded fields can be handled in any number of ways. One way is to use backslashes; to properly parse this data, you should pass the escapechar option:

```
In [164]: data = 'a,b\n"hello, \\"Bob\\", nice to see you",5'

In [165]: print(data)

a,b

"hello, \"Bob\", nice to see you",5

In [166]: pd.read_csv(StringIO(data), escapechar="\\")

Out[166]:

                              a  b

0  hello, "Bob", nice to see you  5
```

Files with fixed width columns

While read_csv() reads delimited data, the read_fwf() function works with data files that have known and fixed column widths. The function parameters to read_fwf are largely the same as read_csv with two extra parameters, and a different usage of the delimiter parameter:

- colspecs: A list of pairs (tuples) giving the extents of the fixed-width fields of each line as half-open intervals (i.e., [from, to[ ). String value ‘infer’ can be used to instruct the parser to try detecting the column specifications from the first 100 rows of the data. Default behavior, if not specified, is to infer.
- widths: A list of field widths which can be used instead of ‘colspecs’ if the intervals are contiguous.
- delimiter: Characters to consider as filler characters in the fixed-width file. Can be used to specify the filler character of the fields if it is not spaces (e.g., ‘~’).

Consider a typical fixed-width data file:

```
In [167]: print(open("bar.csv").read())
```

```
id8141    360.242940   149.910199   11950.7
id1594    444.953632   166.985655   11788.4
id1849    364.136849   183.628767   11806.2
id1230    413.836124   184.375703   11916.8
id1948    502.953953   173.237159   12468.3
```

In order to parse this file into a DataFrame, we simply need to supply the column specifications to the read_fwf function along with the file name:

```
# Column specifications are a list of half-intervals

In [168]: colspecs = [(0, 6), (8, 20), (21, 33), (34, 43)]

In [169]: df = pd.read_fwf("bar.csv", colspecs=colspecs, header=None, index_col=0)

In [170]: df

Out[170]:

                 1           2        3

0

id8141  360.242940  149.910199  11950.7

id1594  444.953632  166.985655  11788.4

id1849  364.136849  183.628767  11806.2

id1230  413.836124  184.375703  11916.8

id1948  502.953953  173.237159  12468.3
```

Note how the parser automatically picks column names X.<column number> when header=None argument is specified. Alternatively, you can supply just the column widths for contiguous columns:

```
# Widths are a list of integers

In [171]: widths = [6, 14, 13, 10]

In [172]: df = pd.read_fwf("bar.csv", widths=widths, header=None)

In [173]: df

Out[173]:

       0         1          2       3

0  id8141  360.242940  149.910199  11950.7

1  id1594  444.953632  166.985655  11788.4

2  id1849  364.136849  183.628767  11806.2

3  id1230  413.836124  184.375703  11916.8

4  id1948  502.953953  173.237159  12468.3
```

The parser will take care of extra white spaces around the columns so it’s ok to have extra separation between the columns in the file.

By default, read_fwf will try to infer the file’s colspecs by using the first 100 rows of the file. It can do it only in cases when the columns are aligned and correctly separated by the provided delimiter (default delimiter is whitespace).

```
In [174]: df = pd.read_fwf("bar.csv", header=None, index_col=0)

In [175]: df

Out[175]:
```

```
              1           2        3
0
id8141  360.242940  149.910199  11950.7
id1594  444.953632  166.985655  11788.4
id1849  364.136849  183.628767  11806.2
id1230  413.836124  184.375703  11916.8
id1948  502.953953  173.237159  12468.3
```

read_fwf supports the dtype parameter for specifying the types of parsed columns to be different from the inferred type.

```
In [176]: pd.read_fwf("bar.csv", header=None, index_col=0).dtypes
Out[176]:
1    float64
2    float64
3    float64
dtype: object

In [177]: pd.read_fwf("bar.csv", header=None, dtype={2: "object"}).dtypes
Out[177]:
0     object
1    float64
2     object
```

```
3    float64

dtype: object
```

Indexes

Files with an “implicit” index column

Consider a file with one less entry in the header than the number of data column:

```
In [178]: print(open("foo.csv").read())

A,B,C

20090101,a,1,2

20090102,b,3,4

20090103,c,4,5
```

In this special case, read_csv assumes that the first column is to be used as the index of the DataFrame:

```
In [179]: pd.read_csv("foo.csv")

Out[179]:

          A  B  C

20090101  a  1  2

20090102  b  3  4

20090103  c  4  5
```

Note that the dates weren’t automatically parsed. In that case you would need to do as before:

```
In [180]: df = pd.read_csv("foo.csv", parse_dates=True)

In [181]: df.index

Out[181]: DatetimeIndex(['2009-01-01', '2009-01-02', '2009-01-03'],
dtype='datetime64[ns]', freq=None)
```

Reading an index with a MultiIndex

Suppose you have data indexed by two columns:

```
In [182]: print(open("data/mindex_ex.csv").read())

year,indiv,zit,xit

1977,"A",1.2,.6

1977,"B",1.5,.5

1977,"C",1.7,.8

1978,"A",.2,.06

1978,"B",.7,.2

1978,"C",.8,.3

1978,"D",.9,.5

1978,"E",1.4,.9

1979,"C",.2,.15

1979,"D",.14,.05

1979,"E",.5,.15

1979,"F",1.2,.5
```

```
1979,"G",3.4,1.9
1979,"H",5.4,2.7
1979,"I",6.4,1.2
```

The index_col argument to read_csv can take a list of column numbers to turn multiple columns into a MultiIndex for the index of the returned object:

```
In [183]: df = pd.read_csv("data/mindex_ex.csv", index_col=[0, 1])

In [184]: df
Out[184]:
            zit   xit
year indiv
1977 A      1.20  0.60
     B      1.50  0.50
     C      1.70  0.80
1978 A      0.20  0.06
     B      0.70  0.20
     C      0.80  0.30
     D      0.90  0.50
     E      1.40  0.90
1979 C      0.20  0.15
     D      0.14  0.05
     E      0.50  0.15
```

```
    F      1.20  0.50
    G      3.40  1.90
    H      5.40  2.70
    I     6.40  1.20
```

```
In [185]: df.loc[1978]
Out[185]:
       zit   xit
indiv
A      0.2  0.06
B      0.7  0.20
C      0.8  0.30
D      0.9  0.50
E      1.4  0.90
```

Reading columns with a MultiIndex

By specifying list of row locations for the header argument, you can read in a MultiIndex for the columns. Specifying non-consecutive rows will skip the intervening rows.

```
In [186]: from pandas._testing import makeCustomDataframe as mkdf
In [187]: df = mkdf(5, 3, r_idx_nlevels=2, c_idx_nlevels=4)
In [188]: df.to_csv("mi.csv")
In [189]: print(open("mi.csv").read())
C0,,C_l0_g0,C_l0_g1,C_l0_g2
C1,,C_l1_g0,C_l1_g1,C_l1_g2
C2,,C_l2_g0,C_l2_g1,C_l2_g2
C3,,C_l3_g0,C_l3_g1,C_l3_g2
R0,R1,,,
R_l0_g0,R_l1_g0,R0C0,R0C1,R0C2
R_l0_g1,R_l1_g1,R1C0,R1C1,R1C2
R_l0_g2,R_l1_g2,R2C0,R2C1,R2C2
R_l0_g3,R_l1_g3,R3C0,R3C1,R3C2
R_l0_g4,R_l1_g4,R4C0,R4C1,R4C2
```

```
In [190]: pd.read_csv("mi.csv", header=[0, 1, 2, 3], index_col=[0, 1])
Out[190]:
C0          C_l0_g0 C_l0_g1 C_l0_g2
C1          C_l1_g0 C_l1_g1 C_l1_g2
C2          C_l2_g0 C_l2_g1 C_l2_g2
C3          C_l3_g0 C_l3_g1 C_l3_g2
R0    R1
```

```
R_l0_g0 R_l1_g0   R0C0   R0C1   R0C2
R_l0_g1 R_l1_g1   R1C0   R1C1   R1C2
R_l0_g2 R_l1_g2   R2C0   R2C1   R2C2
R_l0_g3 R_l1_g3   R3C0   R3C1   R3C2
R_l0_g4 R_l1_g4   R4C0   R4C1   R4C2
```

read_csv is also able to interpret a more common format of multi-columns indices.

```
In [191]: print(open("mi2.csv").read())
,a,a,a,b,c,c
,q,r,s,t,u,v
one,1,2,3,4,5,6
two,7,8,9,10,11,12


In [192]: pd.read_csv("mi2.csv", header=[0, 1], index_col=0)
Out[192]:
     a        b   c
     q  r  s  t   u   v
one  1  2  3  4   5   6
two  7  8  9  10  11  12
```

Note: If an index_col is not specified (e.g. you don’t have an index, or wrote it with df.to_csv(..., index=False), then any names on the columns index will be *lost*.

Automatically “sniffing” the delimiter

read_csv is capable of inferring delimited (not necessarily comma-separated) files, as pandas uses the csv.Sniffer class of the csv module. For this, you have to specify sep=None.

```
In [193]: print(open("tmp2.sv").read())

:0:1:2:3

0:0.4691122999071863:-0.2828633443286633:-1.5090585031735124:-1.1356323710171934

1:1.2121120250208506:-0.17321464905330858:0.11920871129693428:-1.0442359662799567

2:-0.8618489633477999:-2.1045692188948086:-0.4949292740687813:1.071803807037338

3:0.7215551622443669:-0.7067711336300845:-1.0395749851146963:0.27185988554282986

4:-0.42497232978883753:0.567020349793672:0.27623201927771873:-1.0874006912859915

5:-0.6736897080883706:0.1136484096888855:-1.4784265524372235:0.5249876671147047

6:0.4047052186802365:0.5770459859204836:-1.7150020161146375:-1.0392684835147725

7:-0.3706468582364464:-1.1578922506419993:-1.344311812731667:0.8448851414248841

8:1.0757697837155533:-0.10904997528022223:1.6435630703622064:-1.4693879595399115
```

```
9:0.3570205641330908 6:-0.6746001037299882:-1.776903716971867:-0.96
89138124473498
```

```
In [194]: pd.read_csv("tmp2.sv", sep=None, engine="python")
```

```
Out[194]:
```

```
   Unnamed: 0         0         1         2         3
0           0  0.469112 -0.282863 -1.509059 -1.135632
1           1  1.212112 -0.173215  0.119209 -1.044236
2           2 -0.861849 -2.104569 -0.494929  1.071804
3           3  0.721555 -0.706771 -1.039575  0.271860
4           4 -0.424972  0.567020  0.276232 -1.087401
5           5 -0.673690  0.113648 -1.478427  0.524988
6           6  0.404705  0.577046 -1.715002 -1.039268
7           7 -0.370647 -1.157892 -1.344312  0.844885
8           8  1.075770 -0.109050  1.643563 -1.469388
9           9  0.357021 -0.674600 -1.776904 -0.968914
```

Reading multiple files to create a single DataFrame

It’s best to use concat() to combine multiple files. See the cookbook for an example.

Iterating through files chunk by chunk

Suppose you wish to iterate through a (potentially very large) file lazily rather than reading the entire file into memory, such as the following:

```
In [195]: print(open("tmp.sv").read())
```

```
|0|1|2|3

0|0.4691122999071863|-0.2828633443286633|-1.5090585031735124|-1.1356
323710171934

1|1.2121120250208506|-0.17321464905330858|0.11920871129693428|-1.044
2359662799567

2|-0.8618489633477999|-2.1045692188948086|-0.4949292740687813|1.0718
03807037338

3|0.7215551622443669|-0.7067711336300845|-1.0395749851146963|0.2718
5988554282986

4|-0.42497232978883753|0.567020349793672|0.27623201927771873|-1.087
4006912859915

5|-0.6736897080883706|0.1136484096888855|-1.4784265524372235|0.5249
876671147047

6|0.4047052186802365|0.5770459859204836|-1.7150020161146375|-1.0392
684835147725

7|-0.3706468582364464|-1.1578922506419993|-1.344311812731667|0.84488
51414248841

8|1.0757697837155533|-0.10904997528022223|1.6435630703622064|-1.469
3879595399115

9|0.35702056413309086|-0.6746001037299882|-1.776903716971867|-0.9689
138124473498
```

```
In [196]: table = pd.read_csv("tmp.sv", sep="|")

In [197]: table

Out[197]:

  Unnamed: 0        0         1         2         3
```

```
0         0  0.469112 -0.282863 -1.509059 -1.135632
1         1  1.212112 -0.173215  0.119209 -1.044236
2         2 -0.861849 -2.104569 -0.494929  1.071804
3         3  0.721555 -0.706771 -1.039575  0.271860
4         4 -0.424972  0.567020  0.276232 -1.087401
5         5 -0.673690  0.113648 -1.478427  0.524988
6         6  0.404705  0.577046 -1.715002 -1.039268
7         7 -0.370647 -1.157892 -1.344312  0.844885
8         8  1.075770 -0.109050  1.643563 -1.469388
9         9  0.357021 -0.674600 -1.776904 -0.968914
```

By specifying a chunksize to read_csv, the return value will be an iterable object of type TextFileReader:

```
In [198]: with pd.read_csv("tmp.sv", sep="|", chunksize=4) as reader:
   .....:     reader
   .....:     for chunk in reader:
   .....:         print(chunk)
   .....:
   Unnamed: 0         0         1         2         3
0           0  0.469112 -0.282863 -1.509059 -1.135632
1           1  1.212112 -0.173215  0.119209 -1.044236
2           2 -0.861849 -2.104569 -0.494929  1.071804
```

```
3           3  0.721555 -0.706771 -1.039575  0.271860
   Unnamed: 0         0         1         2         3
4           4 -0.424972  0.567020  0.276232 -1.087401
5           5 -0.673690  0.113648 -1.478427  0.524988
6           6  0.404705  0.577046 -1.715002 -1.039268
7           7 -0.370647 -1.157892 -1.344312  0.844885
   Unnamed: 0         0        1         2         3
8           8  1.075770 -0.10905  1.643563 -1.469388
9           9  0.357021 -0.67460 -1.776904 -0.968914
```

Changed in version 1.2: read_csv/json/sas return a context-manager when iterating through a file.

Specifying iterator=True will also return the TextFileReader object:

```
In [199]: with pd.read_csv("tmp.sv", sep="|", iterator=True) as reader:
   .....:     reader.get_chunk(5)
   .....:
```

Specifying the parser engine

Pandas currently supports three engines, the C engine, the python engine, and an experimental pyarrow engine (requires the pyarrow package). In general, the pyarrow engine is fastest on larger workloads and is equivalent in speed to the C engine on most other workloads. The python engine tends to be slower than the pyarrow and C engines on most workloads. However, the pyarrow engine is much less robust than the C engine, which lacks a few features compared to the Python engine.

Where possible, pandas uses the C parser (specified as engine='c'), but it may fall back to Python if C-unsupported options are specified.

Currently, options unsupported by the C and pyarrow engines include:

- sep other than a single character (e.g. regex separators)
- skipfooter
- sep=None with delim_whitespace=False

Specifying any of the above options will produce a ParserWarning unless the python engine is selected explicitly using engine='python'.

Options that are unsupported by the pyarrow engine which are not covered by the list above include:

- **float_precision**
- **chunksize**
- **comment**
- **nrows**
- **thousands**
- **memory_map**
- **dialect**
- **warn_bad_lines**
- **error_bad_lines**
- **on_bad_lines**
- **delim_whitespace**
- **quoting**
- **lineterminator**
- **converters**
- **decimal**

- **iterator**
- **dayfirst**
- **infer_datetime_format**
- **verbose**
- **skipinitialspace**
- **low_memory**

Specifying these options with engine='pyarrow' will raise a ValueError.

Reading/writing remote files

You can pass in a URL to read or write remote files to many of pandas’ IO functions - the following example shows reading a CSV file:

```
df = pd.read_csv("https://download.bls.gov/pub/time.series/cu/cu.item",
sep="\t")
```

New in version 1.3.0.

A custom header can be sent alongside HTTP(s) requests by passing a dictionary of header key value mappings to the storage_options keyword argument as shown below:

```
headers = {"User-Agent": "pandas"}

df = pd.read_csv(

    "https://download.bls.gov/pub/time.series/cu/cu.item",

    sep="\t",

    storage_options=headers

)
```

All URLs which are not local files or HTTP(s) are handled by fsspec, if installed, and its various filesystem implementations (including Amazon S3, Google Cloud, SSH, FTP, webHDFS…). Some of these implementations will require additional packages to be installed, for example S3 URLs require the s3fs library:

```
df = pd.read_json("s3://pandas-test/adatafile.json")
```

When dealing with remote storage systems, you might need extra configuration with environment variables or config files in special locations. For example, to access data in your S3 bucket, you will need to define credentials in one of the several ways listed in the S3Fs documentation. The same is true for several of the storage backends, and you should follow the links at fsimpl1 for implementations built into fsspec and fsimpl2 for those not included in the main fsspec distribution.

You can also pass parameters directly to the backend driver. For example, if you do *not* have S3 credentials, you can still access public data by specifying an anonymous connection, such as

New in version 1.2.0.

```
pd.read_csv(
 "s3://ncei-wcsd-archive/data/processed/SH1305/18kHz/SaKe2013"
 "-D20130523-T080854_to_SaKe2013-D20130523-T085643.csv",
 storage_options={"anon": True},
)
```

fsspec also allows complex URLs, for accessing data in compressed archives, local caching of files, and more. To locally cache the above example, you would modify the call to

```
pd.read_csv(    "simplecache::s3://ncei-wcsd-archive/data/processed/SH1305/18kHz/"    "SaKe2013-D20130523-T080854_to_SaKe2013-D20130523-T085643.csv",

    storage_options={"s3": {"anon": True}},

)
```

where we specify that the “anon” parameter is meant for the “s3” part of the implementation, not to the caching implementation. Note that this caches to a temporary directory for the duration of the session only, but you can also specify a permanent store.

Writing out data

Writing to CSV format

The Series and DataFrame objects have an instance method to_csv which allows storing the contents of the object as a comma-separated-values file. The function takes a number of arguments. Only the first is required.

- path_or_buf: A string path to the file to write or a file object. If a file object it must be opened with newline=''
- sep : Field delimiter for the output file (default “,”)
- na_rep: A string representation of a missing value (default ‘’)
- float_format: Format string for floating point numbers
- columns: Columns to write (default None)
- header: Whether to write out the column names (default True)
- index: whether to write row (index) names (default True)
- index_label: Column label(s) for index column(s) if desired. If None (default), and header and index are True, then the index names are used. (A sequence should be given if the DataFrame uses MultiIndex).
- mode : Python write mode, default ‘w’

- encoding: a string representing the encoding to use if the contents are non-ASCII, for Python versions prior to 3
- line_terminator: Character sequence denoting line end (default os.linesep)
- quoting: Set quoting rules as in csv module (default csv.QUOTE_MINIMAL). Note that if you have set a float_format then floats are converted to strings and csv.QUOTE_NONNUMERIC will treat them as non-numeric
- quotechar: Character used to quote fields (default ‘”’)
- doublequote: Control quoting of quotechar in fields (default True)
- escapechar: Character used to escape sep and quotechar when appropriate (default None)
- chunksize: Number of rows to write at a time
- date_format: Format string for datetime objects

Writing a formatted string

The DataFrame object has an instance method to_string which allows control over the string representation of the object. All arguments are optional:

- buf default None, for example a StringIO object
- columns default None, which columns to write
- col_space default None, minimum width of each column.
- na_rep default NaN, representation of NA value
- formatters default None, a dictionary (by column) of functions each of which takes a single argument and returns a formatted string
- float_format default None, a function which takes a single (float) argument and returns a formatted string; to be applied to floats in the DataFrame.
- sparsify default True, set to False for a DataFrame with a hierarchical index to print every MultiIndex key at each row.
- index_names default True, will print the names of the indices
- index default True, will print the index (ie, row labels)
- header default True, will print the column labels
- justify default left, will print column headers left- or right-justified

The Series object also has a to_string method, but with only the buf, na_rep, float_format arguments. There is also a length argument which, if set to True, will additionally output the length of the Series.

JSON

Read and write JSON format files and strings.

Writing JSON

A Series or DataFrame can be converted to a valid JSON string. Use to_json with optional parameters:

- path_or_buf : the pathname or buffer to write the output This can be None in which case a JSON string is returned
- orient :

Series:

  - default is index
  - allowed values are {split, records, index}
- DataFrame:
  - default is columns
  - allowed values are {split, records, index, columns, values, table}
- The format of the JSON string

split	dict like {index -> [index], columns -> [columns], data -> [values]}
records	list like [{column -> value}, … , {column -> value}]
index	dict like {index -> {column -> value}}
columns	dict like {column -> {index -> value}}
values	just the values array
table	adhering to the JSON Table Schema

- date_format : string, type of date conversion, ‘epoch’ for timestamp, ‘iso’ for ISO8601.
- double_precision : The number of decimal places to use when encoding floating point values, default 10.
- force_ascii : force encoded string to be ASCII, default True.
- date_unit : The time unit to encode to, governs timestamp and ISO8601 precision. One of ‘s’, ‘ms’, ‘us’ or ‘ns’ for seconds, milliseconds, microseconds and nanoseconds respectively. Default ‘ms’.
- default_handler : The handler to call if an object cannot otherwise be converted to a suitable format for JSON. Takes a single argument, which is the object to convert, and returns a serializable object.
- lines : If records orient, then will write each record per line as json.

Note NaN’s, NaT’s and None will be converted to null and datetime objects will be converted based on the date_format and date_unit parameters.

```
In [200]: dfj = pd.DataFrame(np.random.randn(5, 2), columns=list("AB"))

In [201]: json = dfj.to_json()

In [202]: json

Out[202]: '{"A":
{"0":-1.2945235903,"1":0.2766617129,"2":-0.0139597524,"3":-0.00615356
99,"4":0.8957173022},"B":
{"0":0.4137381054,"1":-0.472034511,"2":-0.3625429925,"3":-0.923060654,
"4":0.8052440254}}'
```

Orient options

There are a number of different options for the format of the resulting JSON file / string. Consider the following DataFrame and Series:

```
In [203]: dfjo = pd.DataFrame(
  .....:     dict(A=range(1, 4), B=range(4, 7), C=range(7, 10)),
  .....:     columns=list("ABC"),
  .....:     index=list("xyz"),
  .....: )
  .....:

In [204]: dfjo
Out[204]:
   A  B  C
x  1  4  7
y  2  5  8
z  3  6  9

In [205]: sjo = pd.Series(dict(x=15, y=16, z=17), name="D")

In [206]: sjo
Out[206]:
x    15
y    16
z    17
Name: D, dtype: int64
```

Column oriented (the default for DataFrame) serializes the data as nested JSON objects with column labels acting as the primary index:

```
In [207]: dfjo.to_json(orient="columns")

Out[207]: '{"A":{"x":1,"y":2,"z":3},"B":{"x":4,"y":5,"z":6},"C":
{"x":7,"y":8,"z":9}}'


# Not available for Series
```

Index oriented (the default for Series) similar to column oriented but the index labels are now primary:

```
In [208]: dfjo.to_json(orient="index")

Out[208]: '{"x":{"A":1,"B":4,"C":7},"y":{"A":2,"B":5,"C":8},"z":
{"A":3,"B":6,"C":9}}'

In [209]: sjo.to_json(orient="index")

Out[209]: '{"x":15,"y":16,"z":17}'
```

Record oriented serializes the data to a JSON array of column -> value records, index labels are not included. This is useful for passing DataFrame data to plotting libraries, for example the JavaScript library d3.js:

```
In [210]: dfjo.to_json(orient="records")

Out[210]: '[{"A":1,"B":4,"C":7},{"A":2,"B":5,"C":8},{"A":3,"B":6,"C":9}]'

In [211]: sjo.to_json(orient="records")

Out[211]: '[15,16,17]'
```

Value oriented is a bare-bones option which serializes to nested JSON arrays of values only, column and index labels are not included:

```
In [212]: dfjo.to_json(orient="values")

Out[212]: '[[1,4,7],[2,5,8],[3,6,9]]'
```

```
# Not available for Series
```

Split oriented serializes to a JSON object containing separate entries for values, index and columns. Name is also included for Series:

```
In [213]: dfjo.to_json(orient="split")

Out[213]: '{"columns":["A","B","C"],"index":["x","y","z"],"data":[[1,4,7],
[2,5,8],[3,6,9]]}'

In [214]: sjo.to_json(orient="split")

Out[214]: '{"name":"D","index":["x","y","z"],"data":[15,16,17]}'
```

Table oriented serializes to the JSON Table Schema,

allowing for the preservation of metadata including but not limited to dtypes and index names.

Note

Any orient option that encodes to a JSON object will not preserve the ordering of index and column labels during round-trip serialization. If you wish to preserve label ordering use the split option as it uses ordered containers.

Date handling

Writing in ISO date format:

```
In [215]: dfd = pd.DataFrame(np.random.randn(5, 2), columns=list("AB"))

In [216]: dfd["date"] = pd.Timestamp("20130101")

In [217]: dfd = dfd.sort_index(axis=1, ascending=False)

In [218]: json = dfd.to_json(date_format="iso")

In [219]: json

Out[219]: '{"date":{"0":"2013-01-01T00:00:00.000Z","1":"2013-01-
01T00:00:00.000Z","2":"2013-01-01T00:00:00.000Z","3":"2013-01-
01T00:00:00.000Z","4":"2013-01-01T00:00:00.000Z"},"B":
{"0":2.5656459463,"1":1.3403088498,"2":-0.2261692849,"3":0.8138502857
,"4":-0.8273169356},"A":
{"0":-1.2064117817,"1":1.4312559863,"2":-1.1702987971,"3":0.410834511
2,"4":0.1320031703}}'
```

Writing in ISO date format, with microseconds:

```
In [220]: json = dfd.to_json(date_format="iso", date_unit="us")

In [221]: json

Out[221]: '{"date":{"0":"2013-01-01T00:00:00.000000Z","1":"2013-01-
01T00:00:00.000000Z","2":"2013-01-01T00:00:00.000000Z","3":"2013-01-
01T00:00:00.000000Z","4":"2013-01-01T00:00:00.000000Z"},"B":
{"0":2.5656459463,"1":1.3403088498,"2":-0.2261692849,"3":0.8138502857
,"4":-0.8273169356},"A":
{"0":-1.2064117817,"1":1.4312559863,"2":-1.1702987971,"3":0.410834511
2,"4":0.1320031703}}'
```

Epoch timestamps, in seconds:

```
In [222]: json = dfd.to_json(date_format="epoch", date_unit="s")

In [223]: json

Out[223]: '{"date":
{"0":1356998400,"1":1356998400,"2":1356998400,"3":1356998400,"4":135
6998400},"B":
{"0":2.5656459463,"1":1.3403088498,"2":-0.2261692849,"3":0.8138502857
,"4":-0.8273169356},"A":
{"0":-1.2064117817,"1":1.4312559863,"2":-1.1702987971,"3":0.410834511
2,"4":0.1320031703}}'
```

Writing to a file, with a date index and a date column:

```
In [224]: dfj2 = dfj.copy()

In [225]: dfj2["date"] = pd.Timestamp("20130101")

In [226]: dfj2["ints"] = list(range(5))

In [227]: dfj2["bools"] = True


In [228]: dfj2.index = pd.date_range("20130101", periods=5)

In [229]: dfj2.to_json("test.json")

In [230]: with open("test.json") as fh:

   .....:     print(fh.read())

   .....:

{"A":
{"1356998400000":-1.2945235903,"1357084800000":0.2766617129,"13571
71200000":-0.0139597524,"1357257600000":-0.0061535699,"13573440000
00":0.8957173022},"B":
```

```
{"1356998400000":0.4137381054,"1357084800000":-0.472034511,"135717
1200000":-0.3625429925,"1357257600000":-0.923060654,"1357344000000
":0.8052440254},"date":
{"1356998400000":1356998400000,"1357084800000":1356998400000,"135
7171200000":1356998400000,"1357257600000":1356998400000,"1357344
000000":1356998400000},"ints":
{"1356998400000":0,"1357084800000":1,"1357171200000":2,"1357257600
000":3,"1357344000000":4},"bools":
{"1356998400000":true,"1357084800000":true,"1357171200000":true,"1357
257600000":true,"1357344000000":true}}
```

Fallback behavior

If the JSON serializer cannot handle the container contents directly it will fall back in the following manner:

- if the dtype is unsupported (e.g. np.complex_) then the default_handler, if provided, will be called for each value, otherwise an exception is raised.
- if an object is unsupported it will attempt the following:
  - check if the object has defined a toDict method and call it. A toDict method should return a dict which will then be JSON serialized.
  - invoke the default_handler if one was provided.
  - convert the object to a dict by traversing its contents. However this will often fail with an OverflowError or give unexpected results.

In general the best approach for unsupported objects or dtypes is to provide a default_handler. For example:

```
>>> DataFrame([1.0, 2.0, complex(1.0, 2.0)]).to_json()  # raises

RuntimeError: Unhandled numpy dtype 15
```

can be dealt with by specifying a simple default_handler:

```
In [231]: pd.DataFrame([1.0, 2.0, complex(1.0, 2.0)]).to_json(default_handler=str)
```

```
Out[231]: '{"0":{"0":"(1+0j)","1":"(2+0j)","2":"(1+2j)"}}'
```

Reading JSON

Reading a JSON string to pandas object can take a number of parameters. The parser will try to parse a DataFrame if typ is not supplied or is None. To explicitly force Series parsing, pass typ=series

- filepath_or_buffer : a VALID JSON string or file handle / StringIO.

The string could be a URL. Valid URL schemes include http, ftp, S3, and file. For file URLs, a host is expected. For instance, a local file could be file ://localhost/path/to/table.json

- typ : type of object to recover (series or frame), default ‘frame’
- orient :

Series :

- ○ default is index
- ○ allowed values are {split, records, index}
- DataFrame
- ○ default is columns
- ○ allowed values are {split, records, index, columns, values, table}
- The format of the JSON string

split	dict like {index -> [index], columns -> [columns], data -> [values]}
records	list like [{column -> value}, … , {column -> value}]
index	dict like {index -> {column -> value}}
columns	dict like {column -> {index -> value}}
values	just the values array

table	adhering to the JSON Table Schema

- dtype : if True, infer dtypes, if a dict of column to dtype, then use those, if False, then don’t infer dtypes at all, default is True, apply only to the data.
- convert_axes : boolean, try to convert the axes to the proper dtypes, default is True
- convert_dates : a list of columns to parse for dates; If True, then try to parse date-like columns, default is True.
- keep_default_dates : boolean, default True. If parsing dates, then parse the default date-like columns.
- numpy : direct decoding to NumPy arrays. default is False; Supports numeric data only, although labels may be non-numeric. Also note that the JSON ordering MUST be the same for each term if numpy=True.
- precise_float : boolean, default False. Set to enable usage of higher precision (strtod) function when decoding string to double values. Default (False) is to use fast but less precise builtin functionality.
- date_unit : string, the timestamp unit to detect if converting dates. Default None. By default the timestamp precision will be detected, if this is not desired then pass one of ‘s’, ‘ms’, ‘us’ or ‘ns’ to force timestamp precision to seconds, milliseconds, microseconds or nanoseconds respectively.
- lines : reads file as one json object per line.
- encoding : The encoding to use to decode py3 bytes.
- chunksize : when used in combination with lines=True, return a JsonReader which reads in chunksize lines per iteration.

The parser will raise one of ValueError/TypeError/AssertionError if the JSON is not parseable.

If a non-default orient was used when encoding to JSON be sure to pass the same option here so that decoding produces sensible results, see Orient Options for an overview.

Data conversion

The default of convert_axes=True, dtype=True, and convert_dates=True will try to parse the axes, and all of the data into appropriate types, including dates. If you need to override specific dtypes, pass a dict to dtype. convert_axes should only be set to False if you need to preserve string-like numbers (e.g. ‘1’, ‘2’) in an axes.

Note

Large integer values may be converted to dates if convert_dates=True and the data and / or column labels appear ‘date-like’. The exact threshold depends on the date_unit specified. ‘date-like’ means that the column label meets one of the following criteria:

- it ends with '_at'
- it ends with '_time'
- it begins with 'timestamp'
- it is 'modified'
- it is 'date'

Warning

When reading JSON data, automatic coercing into dtypes has some quirks:

- an index can be reconstructed in a different order from serialization, that is, the returned order is not guaranteed to be the same as before serialization
- a column that was float data will be converted to integer if it can be done safely, e.g. a column of 1.
- bool columns will be converted to integer on reconstruction

Thus there are times where you may want to specify specific dtypes via the dtype keyword argument.

Reading from a JSON string:

```
In [232]: pd.read_json(json)

Out[232]:

        date         B         A

0 2013-01-01  2.565646 -1.206412

1 2013-01-01  1.340309  1.431256

2 2013-01-01 -0.226169 -1.170299

3 2013-01-01  0.813850  0.410835

4 2013-01-01 -0.827317  0.132003
```

Reading from a file:

```
In [233]: pd.read_json("test.json")

Out[233]:

                   A         B       date  ints  bools

2013-01-01 -1.294524  0.413738 2013-01-01     0   True

2013-01-02  0.276662 -0.472035 2013-01-01     1   True

2013-01-03 -0.013960 -0.362543 2013-01-01     2   True

2013-01-04 -0.006154 -0.923061 2013-01-01     3   True

2013-01-05  0.895717  0.805244 2013-01-01     4   True
```

Don’t convert any data (but still convert axes and dates):

```
In [234]: pd.read_json("test.json", dtype=object).dtypes

Out[234]:
```

```
A       object
B       object
date     object
ints     object
bools    object
dtype: object
```

Specify dtypes for conversion:

```
In [235]: pd.read_json("test.json", dtype={"A": "float32", "bools": "int8"}).dtypes
Out[235]:
A              float32
B              float64
date     datetime64[ns]
ints             int64
bools             int8
dtype: object
```

Preserve string indices:

```
In [236]: si = pd.DataFrame(
   .....:     np.zeros((4, 4)), columns=list(range(4)), index=[str(i) for i in range(4)]
   .....: )
   .....:

In [237]: si
Out[237]:
     0    1    2    3
0  0.0  0.0  0.0  0.0
1  0.0  0.0  0.0  0.0
2  0.0  0.0  0.0  0.0
3  0.0  0.0  0.0  0.0

In [238]: si.index
Out[238]: Index(['0', '1', '2', '3'], dtype='object')

In [239]: si.columns
Out[239]: Int64Index([0, 1, 2, 3], dtype='int64')

In [240]: json = si.to_json()

In [241]: sij = pd.read_json(json, convert_axes=False)

In [242]: sij
Out[242]:
   0  1  2  3
0  0  0  0  0
1  0  0  0  0
2  0  0  0  0
3  0  0  0  0

In [243]: sij.index
Out[243]: Index(['0', '1', '2', '3'], dtype='object')
In [244]: sij.columns
Out[244]: Index(['0', '1', '2', '3'], dtype='object')
```

Dates written in nanoseconds need to be read back in nanoseconds:

```
In [245]: json = dfj2.to_json(date_unit="ns")

# Try to parse timestamps as milliseconds -> Won't Work
In [246]: dfju = pd.read_json(json, date_unit="ms")
In [247]: dfju
Out[247]:
                            A         B                 date  ints  bools
1356998400000000000 -1.294524  0.413738  1356998400000000000     0
True
1357084800000000000  0.276662 -0.472035  1356998400000000000     1
True
```

```
1357171200000000000 -0.013960 -0.362543  1356998400000000000     2
True

1357257600000000000 -0.006154 -0.923061  1356998400000000000     3
True

1357344000000000000  0.895717  0.805244  1356998400000000000     4
True
```

```
# Let pandas detect the correct precision

In [248]: dfju = pd.read_json(json)


In [249]: dfju

Out[249]:

                   A         B       date  ints  bools

2013-01-01 -1.294524  0.413738 2013-01-01     0   True

2013-01-02  0.276662 -0.472035 2013-01-01     1   True

2013-01-03 -0.013960 -0.362543 2013-01-01     2   True

2013-01-04 -0.006154 -0.923061 2013-01-01     3   True

2013-01-05  0.895717  0.805244 2013-01-01     4   True
```

```
# Or specify that all timestamps are in nanoseconds

In [250]: dfju = pd.read_json(json, date_unit="ns")

In [251]: dfju

Out[251]:
```

```
                   A         B        date  ints  bools
2013-01-01 -1.294524  0.413738 2013-01-01     0   True
2013-01-02  0.276662 -0.472035 2013-01-01     1   True
2013-01-03 -0.013960 -0.362543 2013-01-01     2   True
2013-01-04 -0.006154 -0.923061 2013-01-01     3   True
2013-01-05  0.895717  0.805244 2013-01-01     4   True
```

The Numpy parameter

Note

This param has been deprecated as of version 1.0.0 and will raise a FutureWarning.

This supports numeric data only. Index and columns labels may be non-numeric, e.g. strings, dates etc.

If numpy=True is passed to read_json an attempt will be made to sniff an appropriate dtype during deserialization and to subsequently decode directly to NumPy arrays, bypassing the need for intermediate Python objects.

This can provide speedups if you are deserialising a large amount of numeric data:

```
In [252]: randfloats = np.random.uniform(-100, 1000, 10000)

In [253]: randfloats.shape = (1000, 10)

In [254]: dffloats = pd.DataFrame(randfloats, columns=list("ABCDEFGHIJ"))

In [255]: jsonfloats = dffloats.to_json()
In [256]: %timeit pd.read_json(jsonfloats)

10.5 ms +- 848 us per loop (mean +- std. dev. of 7 runs, 100 loops each)

In [257]: %timeit pd.read_json(jsonfloats, numpy=True)

6.9 ms +- 216 us per loop (mean +- std. dev. of 7 runs, 100 loops each)
```

The speedup is less noticeable for smaller datasets:

```
In [258]: jsonfloats = dffloats.head(100).to_json()

In [259]: %timeit pd.read_json(jsonfloats)

5.51 ms +- 488 us per loop (mean +- std. dev. of 7 runs, 100 loops each)
```

```
In [260]: %timeit pd.read_json(jsonfloats, numpy=True)

4.82 ms +- 569 us per loop (mean +- std. dev. of 7 runs, 100 loops each)
```

Warning

Direct NumPy decoding makes a number of assumptions and may fail or produce unexpected output if these assumptions are not satisfied:

- data is numeric.
- data is uniform. The dtype is sniffed from the first value decoded. A ValueError may be raised, or incorrect output may be produced if this condition is not satisfied.
- labels are ordered. Labels are only read from the first container, it is assumed that each subsequent row / column has been encoded in the same order. This should be satisfied if the data was encoded using to_json but may not be the case if the JSON is from another source.

Normalization

pandas provides a utility function to take a dict or list of dicts and *normalize* this semi-structured data into a flat table.

```
In [261]: data = [
   .....:     {"id": 1, "name": {"first": "Coleen", "last": "Volk"}},
   .....:     {"name": {"given": "Mark", "family": "Regner"}},
   .....:     {"id": 2, "name": "Faye Raker"},
   .....: ]
   .....:


In [262]: pd.json_normalize(data)
Out[262]:
    id name.first name.last name.given name.family        name
0  1.0     Coleen      Volk        NaN         NaN         NaN
1  NaN        NaN       NaN       Mark      Regner         NaN
2  2.0        NaN       NaN        NaN         NaN  Faye Raker
```

```
In [263]: data = [
   .....:     {
   .....:         "state": "Florida",
   .....:         "shortname": "FL",
   .....:         "info": {"governor": "Rick Scott"},
   .....:         "county": [
   .....:             {"name": "Dade", "population": 12345},
   .....:             {"name": "Broward", "population": 40000},
   .....:             {"name": "Palm Beach", "population": 60000},
   .....:         ],
   .....:     },
   .....:     {
   .....:         "state": "Ohio",
   .....:         "shortname": "OH",
   .....:         "info": {"governor": "John Kasich"},
   .....:         "county": [
   .....:             {"name": "Summit", "population": 1234},
   .....:             {"name": "Cuyahoga", "population": 1337},
   .....:         ],
   .....:     },
   .....: ]
   .....:

In [264]: pd.json_normalize(data, "county", ["state", "shortname", ["info",
"governor"]])

Out[264]:

         name  population    state shortname info.governor
```

```
0       Dade      12345  Florida       FL    Rick Scott
1     Broward      40000  Florida       FL    Rick Scott
2  Palm Beach      60000  Florida       FL    Rick Scott
3      Summit      1234     Ohio       OH   John Kasich
4    Cuyahoga      1337     Ohio       OH   John Kasich
```

The max_level parameter provides more control over which level to end normalization. With max_level=1 the following snippet normalizes until 1st nesting level of the provided dict.

```
In [265]: data = [
   .....:     {
   .....:         "CreatedBy": {"Name": "User001"},
   .....:         "Lookup": {
   .....:             "TextField": "Some text",
   .....:             "UserField": {"Id": "ID001", "Name": "Name001"},
   .....:         },
   .....:         "Image": {"a": "b"},
   .....:     }
   .....: ]
   .....:


In [266]: pd.json_normalize(data, max_level=1)

Out[266]:
  CreatedBy.Name Lookup.TextField                Lookup.UserField Image.a
0        User001        Some text  {'Id': 'ID001', 'Name': 'Name001'}       b
```

Line delimited json

pandas is able to read and write line-delimited json files that are common in data processing pipelines using Hadoop or Spark.

For line-delimited json files, pandas can also return an iterator which reads in chunksize lines at a time. This can be useful for large files or to read from a stream.

```
In [267]: jsonl = """
   .....:     {"a": 1, "b": 2}
   .....:     {"a": 3, "b": 4}
   .....: """
   .....:

In [268]: df = pd.read_json(jsonl, lines=True)

In [269]: df
Out[269]:
   a  b
0  1  2
1  3  4
```

```
In [270]: df.to_json(orient="records", lines=True)

Out[270]: '{"a":1,"b":2}\n{"a":3,"b":4}\n'


# reader is an iterator that returns ``chunksize`` lines each iteration

In [271]: with pd.read_json(StringIO(jsonl), lines=True, chunksize=1) as reader:

   .....:     reader

   .....:     for chunk in reader:

   .....:         print(chunk)

   .....:
```

Empty DataFrame

```
Columns: []

Index: []

   a  b

0  1  2

   a  b

1  3  4
```

Table schema

Table Schema is a spec for describing tabular datasets as a JSON object. The JSON includes information on the field names, types, and other attributes. You can use the orient table to build a JSON string with two fields, schema and data.

```
In [272]: df = pd.DataFrame(
   .....:     {
   .....:         "A": [1, 2, 3],
   .....:         "B": ["a", "b", "c"],
   .....:         "C": pd.date_range("2016-01-01", freq="d", periods=3),
   .....:     },
   .....:     index=pd.Index(range(3), name="idx"),
   .....: )
   .....:

In [273]: df
Out[273]:
     A  B          C
idx
0    1  a 2016-01-01
1    2  b 2016-01-02
2    3  c 2016-01-03

In [274]: df.to_json(orient="table", date_format="iso")
Out[274]: '{"schema":{"fields":[{"name":"idx","type":"integer"},
{"name":"A","type":"integer"},{"name":"B","type":"string"},
{"name":"C","type":"datetime"}],"primaryKey":["idx"],"pandas_version":"1.4.0"},"data":[{"idx":0,"A":1,"B":"a","C":"2016-01-
01T00:00:00.000Z"},{"idx":1,"A":2,"B":"b","C":"2016-01-
02T00:00:00.000Z"},{"idx":2,"A":3,"B":"c","C":"2016-01-
03T00:00:00.000Z"}]}'
```

The schema field contains the fields key, which itself contains a list of column name to type pairs, including the Index or MultiIndex (see below for a list of types). The schema field also contains a primaryKey field if the (Multi)index is unique.

The second field, data, contains the serialized data with the records orient. The index is included, and any datetimes are ISO 8601 formatted, as required by the Table Schema spec.

The full list of types supported are described in the Table Schema spec. This table shows the mapping from pandas types:

pandas type	Table Schema type
int64	integer
float64	number
bool	boolean
datetime64[ns]	datetime
timedelta64[ns]	duration
categorical	any
object	str

A few notes on the generated table schema:

- The schema object contains a pandas_version field. This contains the version of pandas’ dialect of the schema, and will be incremented with each revision.

All dates are converted to UTC when serializing. Even timezone naive values, which are treated as UTC with an offset of 0.

```
In [275]: from pandas.io.json import build_table_schema

In [276]: s = pd.Series(pd.date_range("2016", periods=4))

In [277]: build_table_schema(s)

Out[277]:

{'fields': [{'name': 'index', 'type': 'integer'},

  {'name': 'values', 'type': 'datetime'}],

'primaryKey': ['index'],

'pandas_version': '1.4.0'}
```

-

datetimes with a timezone (before serializing), include an additional field tz with the time zone name (e.g. 'US/Central').

```
In [278]: s_tz = pd.Series(pd.date_range("2016", periods=12, tz="US/Cen-
tral"))

In [279]: build_table_schema(s_tz)

Out[279]:

{'fields': [{'name': 'index', 'type': 'integer'},

  {'name': 'values', 'type': 'datetime', 'tz': 'US/Central'}],

'primaryKey': ['index'],

'pandas_version': '1.4.0'}
```

●

Periods are converted to timestamps before serialization, and so have the same behavior of being converted to UTC. In addition, periods will contain and additional field freq with the period’s frequency, e.g. 'A-DEC'.

```
In [280]: s_per = pd.Series(1, index=pd.period_range("2016", freq="A-DEC", periods=4))

In [281]: build_table_schema(s_per)

Out[281]:

{'fields': [{'name': 'index', 'type': 'datetime', 'freq': 'A-DEC'},

  {'name': 'values', 'type': 'integer'}],

'primaryKey': ['index'],

'pandas_version': '1.4.0'}
```

●

Categoricals use the any type and an enum constraint listing the set of possible values. Additionally, an ordered field is included:

```
In [282]: s_cat = pd.Series(pd.Categorical(["a", "b", "a"]))

In [283]: build_table_schema(s_cat)

Out[283]:

{'fields': [{'name': 'index', 'type': 'integer'},

  {'name': 'values',

   'type': 'any',

   'constraints': {'enum': ['a', 'b']},

   'ordered': False}],
'primaryKey': ['index'],

'pandas_version': '1.4.0'}
```

●

A primaryKey field, containing an array of labels, is included *if the index is unique*:

```
In [284]: s_dupe = pd.Series([1, 2], index=[1, 1])

In [285]: build_table_schema(s_dupe)

Out[285]:

{'fields': [{'name': 'index', 'type': 'integer'},

  {'name': 'values', 'type': 'integer'}],

'pandas_version': '1.4.0'}
```

●

The primaryKey behavior is the same with MultiIndexes, but in this case the primaryKey is an array:

```
In [286]: s_multi = pd.Series(1, index=pd.MultiIndex.from_product([("a", "b"), (0, 1)]))

In [287]: build_table_schema(s_multi)

Out[287]:

{'fields': [{'name': 'level_0', 'type': 'string'},

  {'name': 'level_1', 'type': 'integer'},

  {'name': 'values', 'type': 'integer'}],

'primaryKey': FrozenList(['level_0', 'level_1']),

'pandas_version': '1.4.0'}
```

- The default naming roughly follows these rules:
  - For series, the object.name is used. If that’s none, then the name is values
  - For DataFrames, the stringified version of the column name is used
  - For Index (not MultiIndex), index.name is used, with a fallback to index if that is None.
  - For MultiIndex, mi.names is used. If any level has no name, then level_<i> is used.

read_json also accepts orient='table' as an argument. This allows for the preservation of metadata such as dtypes and index names in a round-trippable manner.

```
In [288]: df = pd.DataFrame(
   .....:     {
   .....:         "foo": [1, 2, 3, 4],
   .....:         "bar": ["a", "b", "c", "d"],
   .....:         "baz": pd.date_range("2018-01-01", freq="d", periods=4),
   .....:         "qux": pd.Categorical(["a", "b", "c", "c"]),
   .....:     },
   .....:     index=pd.Index(range(4), name="idx"),
   .....: )
   .....:


In [289]: df

Out[289]:
     foo bar        baz qux
idx
0      1   a 2018-01-01   a
1      2   b 2018-01-02   b
2      3   c 2018-01-03   c
3      4   d 2018-01-04   c

In [290]: df.dtypes
Out[290]:
foo             int64
bar            object
baz    datetime64[ns]
qux          category
dtype: object

In [291]: df.to_json("test.json", orient="table")

In [292]: new_df = pd.read_json("test.json", orient="table")

In [293]: new_df
Out[293]:
     foo bar        baz qux
```

```
idx
0      1   a 2018-01-01   a
1      2   b 2018-01-02   b
2      3   c 2018-01-03   c
3      4   d 2018-01-04   c
```

```
In [294]: new_df.dtypes
Out[294]:
foo             int64
bar            object
baz    datetime64[ns]
qux          category
dtype: object
```

Please note that the literal string 'index' as the name of an Index is not round-trippable, nor are any names beginning with 'level_' within a MultiIndex. These are used by default in DataFrame.to_json() to indicate missing values and the subsequent read cannot distinguish the intent.

```
In [295]: df.index.name = "index"

In [296]: df.to_json("test.json", orient="table")

In [297]: new_df = pd.read_json("test.json", orient="table")

In [298]: print(new_df.index.name)

None
```

When using orient='table' along with user-defined ExtensionArray, the generated schema will contain an additional extDtype key in the respective fields element. This extra key is not standard but does enable JSON roundtrips for extension types (e.g. read_json(df.to_json(orient="table"), orient="table")).

The extDtype key carries the name of the extension, if you have properly registered the ExtensionDtype, pandas will use said name to perform a lookup into the registry and re-convert the serialized data into your custom dtype.

HTML

Reading HTML content

Warning

We highly encourage you to read the HTML Table Parsing gotchas below regarding the issues surrounding the BeautifulSoup4/html5lib/lxml parsers.

The top-level read_html() function can accept an HTML string/file/URL and will parse HTML tables into list of pandas DataFrames. Let’s look at a few examples.

Note

read_html returns a list of DataFrame objects, even if there is only a single table contained in the HTML content.

Read a URL with no options:

```
In [299]: url = "https://www.fdic.gov/resources/resolutions/bank-failures/failed-bank-list"
In [300]: dfs = pd.read_html(url)
```

```
In [301]: dfs
```

```
Out[301]:

[                    Bank NameBank          CityCity StateSt  ...           Acquiring
InstitutionAI Closing DateClosing FundFund

0                  Almena State Bank             Almena      KS  ...
Equity Bank    October 23, 2020     10538

1          First City Bank of Florida  Fort Walton Beach      FL  ...           United
Fidelity Bank, fsb    October 16, 2020     10537

2                The First State Bank      Barboursville      WV  ...
MVB Bank, Inc.       April 3, 2020     10536

3                 Ericson State Bank            Ericson      NE  ...          Farmers and
Merchants Bank   February 14, 2020     10535

4     City National Bank of New Jersey            Newark      NJ  ...
Industrial Bank    November 1, 2019     10534

..                              ...               ...     ...  ...                          ...
...       ...

558                 Superior Bank, FSB           Hinsdale      IL  ...              Superior
Federal, FSB       July 27, 2001      6004

559                Malta National Bank              Malta      OH  ...                  North
Valley Bank         May 3, 2001      4648

560    First Alliance Bank & Trust Co.         Manchester      NH  ...  Southern
New Hampshire Bank & Trust    February 2, 2001      4647

561  National State Bank of Metropolis         Metropolis      IL  ...             Ban-
terra Bank of Marion   December 14, 2000      4646
562              Bank of Honolulu         Honolulu     HI  ...                Bank of the Orient    October 13, 2000     4645

[563 rows x 7 columns]]
```

Note

The data from the above URL changes every Monday so the resulting data above and the data below may be slightly different.

Read in the content of the file from the above URL and pass it to read_html as a string:

```
In [302]: with open(file_path, "r") as f:
   .....:     dfs = pd.read_html(f.read())
   .....:

In [303]: dfs
Out[303]:
[                         Bank Name         City  ...       Closing Date       Updated Date
0    Banks of Wisconsin d/b/a Bank of Kenosha       Kenosha  ...       May 31, 2013       May 31, 2013
1                    Central Arizona Bank    Scottsdale  ...       May 14, 2013       May 20, 2013
2                          Sunrise Bank      Valdosta  ...       May 10, 2013       May 21, 2013
3                  Pisgah Community Bank     Asheville  ...      May 10, 2013
May 14, 2013

4                   Douglas County Bank  Douglasville  ...     April 26, 2013
May 16, 2013

..                                  ...           ...  ...                ...              ...

501                   Superior Bank, FSB      Hinsdale  ...      July 27, 2001
June 5, 2012

502                  Malta National Bank         Malta  ...        May 3, 2001  November 18, 2002

503          First Alliance Bank & Trust Co.    Manchester  ...   February 2,
2001  February 18, 2003

504        National State Bank of Metropolis    Metropolis  ... December 14,
2000     March 17, 2005

505                     Bank of Honolulu      Honolulu  ...  October 13, 2000
March 17, 2005


[506 rows x 7 columns]]
```

You can even pass in an instance of StringIO if you so desire:

```
In [304]: with open(file_path, "r") as f:
   .....:     sio = StringIO(f.read())
   .....:


In [305]: dfs = pd.read_html(sio)
In [306]: dfs

Out[306]:

[                          Bank Name        City  ...     Closing Date     Updated Date

0    Banks of Wisconsin d/b/a Bank of Kenosha     Kenosha  ...     May 31, 2013     May 31, 2013

1                   Central Arizona Bank  Scottsdale  ...     May 14, 2013     May 20, 2013

2                        Sunrise Bank    Valdosta  ...     May 10, 2013     May 21, 2013

3                   Pisgah Community Bank   Asheville  ...     May 10, 2013     May 14, 2013

4                    Douglas County Bank  Douglasville  ...   April 26, 2013     May 16, 2013

..                                  ...         ...  ...              ...              ...

501                    Superior Bank, FSB    Hinsdale  ...    July 27, 2001     June 5, 2012

502                   Malta National Bank       Malta  ...      May 3, 2001  November 18, 2002

503          First Alliance Bank & Trust Co.  Manchester  ...  February 2, 2001  February 18, 2003

504        National State Bank of Metropolis  Metropolis  ...  December 14, 2000    March 17, 2005

505                     Bank of Honolulu    Honolulu  ...  October 13, 2000    March 17, 2005
[506 rows x 7 columns]]
```

Note

The following examples are not run by the IPython evaluator due to the fact that having so many network-accessing functions slows down the documentation build. If you spot an error or an example that doesn’t run, please do not hesitate to report it over on pandas GitHub issues page.

Read a URL and match a table that contains specific text:

```
match = "Metcalf Bank"
```

```
df_list = pd.read_html(url, match=match)
```

Specify a header row (by default <th> or <td> elements located within a <thead> are used to form the column index, if multiple rows are contained within <thead> then a MultiIndex is created); if specified, the header row is taken from the data minus the parsed header elements (<th> elements).

```
dfs = pd.read_html(url, header=0)
```

Specify an index column:

```
dfs = pd.read_html(url, index_col=0)
```

Specify a number of rows to skip:

```
dfs = pd.read_html(url, skiprows=0)
```

Specify a number of rows to skip using a list (range works as well):

```
dfs = pd.read_html(url, skiprows=range(2))
```

Specify an HTML attribute:

```
dfs1 = pd.read_html(url, attrs={"id": "table"})

dfs2 = pd.read_html(url, attrs={"class": "sortable"})

print(np.array_equal(dfs1[0], dfs2[0]))  # Should be True
```

Specify values that should be converted to NaN:

```
dfs = pd.read_html(url, na_values=["No Acquirer"])
```

Specify whether to keep the default set of NaN values:

```
dfs = pd.read_html(url, keep_default_na=False)
```

Specify converters for columns. This is useful for numerical text data that has leading zeros. By default columns that are numerical are cast to numeric types and the leading zeros are lost. To avoid this, we can convert these columns to strings.

```
url_mcc = "https://en.wikipedia.org/wiki/Mobile_country_code"

dfs = pd.read_html(

  url_mcc,

  match="Telekom Albania",

  header=0,
    converters={"MNC": str},
)
```

Use some combination of the above:

```
dfs = pd.read_html(url, match="Metcalf Bank", index_col=0)
```

Read in pandas to_html output (with some loss of floating point precision):

```
df = pd.DataFrame(np.random.randn(2, 2))
s = df.to_html(float_format="{0:.40g}".format)
dfin = pd.read_html(s, index_col=0)
```

The lxml backend will raise an error on a failed parse if that is the only parser you provide. If you only have a single parser you can provide just a string, but it is considered good practice to pass a list with one string if, for example, the function expects a sequence of strings. You may use:

```
dfs = pd.read_html(url, "Metcalf Bank", index_col=0, flavor=["lxml"])
```

Or you could pass flavor='lxml' without a list:

```
dfs = pd.read_html(url, "Metcalf Bank", index_col=0, flavor="lxml")
```

However, if you have bs4 and html5lib installed and pass None or ['lxml', 'bs4'] then the parse will most likely succeed. Note that *as soon as a parse succeeds, the function will return*.

```
dfs = pd.read_html(url, "Metcalf Bank", index_col=0, flavor=["lxml", "bs4"])
```

Writing to HTML files

DataFrame objects have an instance method to_html which renders the contents of the DataFrame as an HTML table. The function arguments are as in the method to_string described above.

Note

Not all of the possible options for DataFrame.to_html are shown here for brevity's sake. See to_html() for the full set of options.

```
In [307]: df = pd.DataFrame(np.random.randn(2, 2))

In [308]: df

Out[308]:

          0         1

0 -0.184744  0.496971

1 -0.856240  1.857977


In [309]: print(df.to_html())  # raw html

<table border="1" class="dataframe">

  <thead>

    <tr style="text-align: right;">

      <th></th>

      <th>0</th>

      <th>1</th>
```

```
   </tr>
  </thead>
  <tbody>
   <tr>
    <th>0</th>
    <td>-0.184744</td>
    <td>0.496971</td>
   </tr>
   <tr>
    <th>1</th>
    <td>-0.856240</td>
    <td>1.857977</td>
   </tr>
  </tbody>
</table>
```

HTML:

	0	1
0	-0.184744	0.496971
1	-0.856240	1.857977

The columns argument will limit the columns shown:

```
In [310]: print(df.to_html(columns=[0]))
<table border="1" class="dataframe">
  <thead>
    <tr style="text-align: right;">
      <th></th>
      <th>0</th>
    </tr>
  </thead>
  <tbody>
    <tr>
      <th>0</th>
      <td>-0.184744</td>
    </tr>
    <tr>
      <th>1</th>
      <td>-0.856240</td>
    </tr>
  </tbody>
</table>
```

HTML:

	0

0	-0.184744
1	-0.856240

float_format takes a Python callable to control the precision of floating point values:

```
In [311]: print(df.to_html(float_format="{0:.10f}".format))

<table border="1" class="dataframe">

  <thead>

    <tr style="text-align: right;">

      <th></th>

      <th>0</th>

      <th>1</th>

    </tr>

  </thead>

  <tbody>

    <tr>

      <th>0</th>

      <td>-0.1847438576</td>

      <td>0.4969711327</td>

    </tr>

    <tr>

      <th>1</th>
      <td>-0.8562396763</td>
      <td>1.8579766508</td>
    </tr>
  </tbody>
</table>
```

HTML:

	0	1
0	-0.1847438576	0.4969711327
1	-0.8562396763	1.8579766508

bold_rows will make the row labels bold by default, but you can turn that off:

```
In [312]: print(df.to_html(bold_rows=False))
<table border="1" class="dataframe">
  <thead>
    <tr style="text-align: right;">
      <th></th>
      <th>0</th>
      <th>1</th>
    </tr>
  </thead>
  <tbody>
```

```
    <tr>
      <td>0</td>
      <td>-0.184744</td>
      <td>0.496971</td>
    </tr>
    <tr>
      <td>1</td>
      <td>-0.856240</td>
      <td>1.857977</td>
    </tr>
  </tbody>
</table>
```

	0	1
0	-0.184744	0.496971
1	-0.856240	1.857977

The classes argument provides the ability to give the resulting HTML table CSS classes. Note that these classes are *appended* to the existing 'dataframe' class.

```
In [313]: print(df.to_html(classes=["awesome_table_class", "even_more_awesome_class"]))
```

```
<table border="1" class="dataframe awesome_table_class even_more_awesome_class">
```

```
  <thead>
    <tr style="text-align: right;">
      <th></th>
      <th>0</th>
      <th>1</th>
    </tr>
  </thead>
  <tbody>
    <tr>
      <th>0</th>
      <td>-0.184744</td>
      <td>0.496971</td>
    </tr>
    <tr>
      <th>1</th>
      <td>-0.856240</td>
      <td>1.857977</td>
    </tr>
  </tbody>
</table>
```

The render_links argument provides the ability to add hyperlinks to cells that contain URLs.

```
In [314]: url_df = pd.DataFrame(
   .....:     {
   .....:         "name": ["Python", "pandas"],
   .....:         "url": ["https://www.python.org/", "https://pandas.pydata.org"],
   .....:     }
   .....: )
   .....:

In [315]: print(url_df.to_html(render_links=True))
<table border="1" class="dataframe">
  <thead>
    <tr style="text-align: right;">
      <th></th>
      <th>name</th>
      <th>url</th>
    </tr>
  </thead>
  <tbody>
    <tr>
      <th>0</th>
      <td>Python</td>
      <td><a href="https://www.python.org/" target="_blank">https://www.python.org/</a></td>
    </tr>
    <tr>
      <th>1</th>
      <td>pandas</td>
      <td><a href="https://pandas.pydata.org" target="_blank">https://pandas.pydata.org</a></td>
    </tr>
  </tbody>
</table>
```

HTML:

	name	url
0	Python	https://www.python.org/
1	pandas	https://pandas.pydata.org

Finally, the escape argument allows you to control whether the “<”, “>” and “&” characters escaped in the resulting HTML (by default it is True). So to get the HTML without escaped characters pass escape=False

```
In [316]: df = pd.DataFrame({"a": list("&<>"), "b": np.random.randn(3)})
```

Escaped:

```
In [317]: print(df.to_html())
<table border="1" class="dataframe">
  <thead>
    <tr style="text-align: right;">
      <th></th>
      <th>a</th>
      <th>b</th>
    </tr>
  </thead>
  <tbody>
    <tr>
      <th>0</th>
      <td>&amp;</td>
      <td>-0.474063</td>
    </tr>
    <tr>
      <th>1</th>
      <td>&lt;</td>
      <td>-0.230305</td>
    </tr>
    <tr>
```

```
      <th>2</th>
      <td>&gt;</td>
      <td>-0.400654</td>
    </tr>
  </tbody>
</table>
```

	a	b
0	&	-0.474063
1	<	-0.230305
2	>	-0.400654

Not escaped:

```
In [318]: print(df.to_html(escape=False))
<table border="1" class="dataframe">
  <thead>
    <tr style="text-align: right;">
      <th></th>
      <th>a</th>
      <th>b</th>
    </tr>
  </thead>
  <tbody>
```

```
    <tr>
      <th>0</th>
      <td>&</td>
      <td>-0.474063</td>
    </tr>
    <tr>
      <th>1</th>
      <td><</td>
      <td>-0.230305</td>
    </tr>
    <tr>
      <th>2</th>
      <td>></td>
      <td>-0.400654</td>
    </tr>
  </tbody>
</table>
```

	a	b
0	&	-0.474063
1	<	-0.230305

2	>	-0.400654

Note

Some browsers may not show a difference in the rendering of the previous two HTML tables.

HTML Table Parsing Gotchas

There are some versioning issues surrounding the libraries that are used to parse HTML tables in the top-level pandas io function read_html.

Issues with lxml

- Benefits
  - lxml is very fast.
  - lxml requires Cython to install correctly.
- Drawbacks
  - lxml does *not* make any guarantees about the results of its parse *unless* it is given strictly valid markup.
  - In light of the above, we have chosen to allow you, the user, to use the lxml backend, but this backend will use html5lib if lxml fails to parse
  - It is therefore *highly recommended* that you install both BeautifulSoup4 and html5lib, so that you will still get a valid result (provided everything else is valid) even if lxml fails.

Issues with BeautifulSoup4 using lxml as a backend

- The above issues hold here as well since BeautifulSoup4 is essentially just a wrapper around a parser backend.

Issues with BeautifulSoup4 using html5lib as a backend

- Benefits
  - html5lib is far more lenient than lxml and consequently deals with *real-life markup* in a much saner way rather than just, e.g., dropping an ele-

ment without notifying you.

○ html5lib *generates valid HTML5 markup from invalid markup automatically*. This is extremely important for parsing HTML tables, since it guarantees a valid document. However, that does NOT mean that it is “correct”, since the process of fixing markup does not have a single definition.

○ html5lib is pure Python and requires no additional build steps beyond its own installation.

- Drawbacks

○ The biggest drawback to using html5lib is that it is slow as molasses. However consider the fact that many tables on the web are not big enough for the parsing algorithm runtime to matter. It is more likely that the bottleneck will be in the process of reading the raw text from the URL over the web, i.e., IO (input-output). For very large tables, this might not be true.

Data Cleaning

Data cleaning means fixing bad data in your data set.

Bad data could be:

- Empty cells
- Data in wrong format
- Wrong data

- Duplicates

In this tutorial you will learn how to deal with all of them.

Our Data Set

In the next chapters we will use this data set:

```
   Duration          Date  Pulse  Maxpulse  Calories
0        60  '2020/12/01'    110       130     409.1
1        60  '2020/12/02'    117       145     479.0
2        60  '2020/12/03'    103       135     340.0
3        45  '2020/12/04'    109       175     282.4
4        45  '2020/12/05'    117       148     406.0
5        60  '2020/12/06'    102       127     300.0
6        60  '2020/12/07'    110       136     374.0
7       450  '2020/12/08'    104       134     253.3
8        30  '2020/12/09'    109       133     195.1
9        60  '2020/12/10'     98       124     269.0
10       60  '2020/12/11'    103       147     329.3
11       60  '2020/12/12'    100       120     250.7
12       60  '2020/12/12'    100       120     250.7
13       60  '2020/12/13'    106       128     345.3
14       60  '2020/12/14'    104       132     379.3
15       60  '2020/12/15'     98       123     275.0
```

```
16     60  '2020/12/16'    98     120    215.2
17     60  '2020/12/17'   100     120    300.0
18     45  '2020/12/18'    90     112      NaN
19     60  '2020/12/19'   103     123    323.0
20     45  '2020/12/20'    97     125    243.0
21     60  '2020/12/21'   108     131    364.2
22     45          NaN   100     119    282.0
23     60  '2020/12/23'   130     101    300.0
24     45  '2020/12/24'   105     132    246.0
25     60  '2020/12/25'   102     126    334.5
26     60    2020/12/26   100     120    250.0
27     60  '2020/12/27'    92     118    241.0
28     60  '2020/12/28'   103     132      NaN
29     60  '2020/12/29'   100     132    280.0
30     60  '2020/12/30'   102     129    380.3
31     60  '2020/12/31'    92     115    243.0
```

The data set contains some empty cells ("Date" in row 22, and "Calories" in row 18 and 28).

The data set contains wrong format ("Date" in row 26).

The data set contains wrong data ("Duration" in row 7).

The data set contains duplicates (row 11 and 12).

Empty Cells

Empty cells can potentially give you a wrong result when you analyze data.

Remove Rows

One way to deal with empty cells is to remove rows that contain empty cells.

This is usually OK, since data sets can be very big, and removing a few rows will not have a big impact on the result.

Example

Return a new Data Frame with no empty cells:

```
import pandas as pd

df = pd.read_csv('data.csv')

new_df = df.dropna()

print(new_df.to_string())
```

In our cleaning examples we will be using a CSV file called 'dirtydata.csv'.

Download dirtydata.csv. or Open dirtydata.csv

Note: By default, the dropna() method returns a *new* DataFrame, and will not change the original.

If you want to change the original DataFrame, use the inplace = True argument:

Example

Remove all rows with NULL values:

```
import pandas as pd
```

```
df = pd.read_csv('data.csv')

df.dropna(inplace = True)

print(df.to_string())
```

Note: Now, the dropna(inplace = True) will NOT return a new DataFrame, but it will remove all rows containg NULL values from the original DataFrame.

Replace Empty Values

Another way of dealing with empty cells is to insert a *new* value instead.

This way you do not have to delete entire rows just because of some empty cells.

The fillna() method allows us to replace empty cells with a value:

Example

Replace NULL values with the number 130:

```
import pandas as pd

df = pd.read_csv('data.csv')

df.fillna(130, inplace = True)
```

Replace Only For Specified Columns

The example above replaces all empty cells in the whole Data Frame.

To only replace empty values for one column,

specify the *column name* for the DataFrame:

Example

Replace NULL values in the "Calories" columns with the number 130:

```
import pandas as pd

df = pd.read_csv('data.csv')

df["Calories"].fillna(130, inplace = True)
```

Replace Using Mean, Median, or Mode

A common way to replace empty cells, is to calculate the mean, median or mode value of the column.

Pandas uses the mean() median() and mode() methods to calculate the respective values for a specified column:

Example

Calculate the MEAN, and replace any empty values with it:

```
import pandas as pd

df = pd.read_csv('data.csv')

x = df["Calories"].mean()

df["Calories"].fillna(x, inplace = True)
```

Mean = the average value (the sum of all values divided by number of values).

Example

Calculate the MEDIAN, and replace any empty values with it:

```
import pandas as pd

df = pd.read_csv('data.csv')

x = df["Calories"].median()

df["Calories"].fillna(x, inplace = True)
```

Median = the value in the middle, after you have sorted all values ascending.

Example

Calculate the MODE, and replace any empty values with it:

```
import pandas as pd

df = pd.read_csv('data.csv')

x = df["Calories"].mode()[0]

df["Calories"].fillna(x, inplace = True)
```

Mode = the value that appears most frequently.

Pandas - Cleaning Data of Wrong Format

Data of Wrong Format

Cells with data of wrong format can make it difficult, or even impossible, to analyze data.

To fix it, you have two options: remove the rows, or convert all cells in the columns into the same format.

Convert Into a Correct Format

In our Data Frame, we have two cells with the wrong format. Check out row 22 and 26, the 'Date' column should be a string that represents a date:

```
  Duration          Date  Pulse  Maxpulse  Calories

0        60  '2020/12/01'    110       130     409.1

1        60  '2020/12/02'    117       145     479.0

2        60  '2020/12/03'    103       135     340.0

3        45  '2020/12/04'    109       175     282.4

4        45  '2020/12/05'    117       148     406.0

5        60  '2020/12/06'    102       127     300.0

6        60  '2020/12/07'    110       136     374.0

7       450  '2020/12/08'    104       134     253.3

8        30  '2020/12/09'    109       133     195.1

9        60  '2020/12/10'     98       124     269.0

10       60  '2020/12/11'    103       147     329.3
```

```
11      60  '2020/12/12'   100      120    250.7
12      60  '2020/12/12'   100      120    250.7
13      60  '2020/12/13'   106      128    345.3
14      60  '2020/12/14'   104      132    379.3
15      60  '2020/12/15'    98      123    275.0
16      60  '2020/12/16'    98      120    215.2
17      60  '2020/12/17'   100      120    300.0
18      45  '2020/12/18'    90      112      NaN
19      60  '2020/12/19'   103      123    323.0
20      45  '2020/12/20'    97      125    243.0
21      60  '2020/12/21'   108      131    364.2
22      45          NaN   100      119    282.0
23      60  '2020/12/23'   130      101    300.0
24      45  '2020/12/24'   105      132    246.0
25      60  '2020/12/25'   102      126    334.5
26      60      20201226   100      120    250.0
27      60  '2020/12/27'    92      118    241.0
28      60  '2020/12/28'   103      132      NaN
29      60  '2020/12/29'   100      132    280.0
30      60  '2020/12/30'   102      129    380.3
31      60  '2020/12/31'    92      115    243.0
```

Let's try to convert all cells in the 'Date' column into dates.

Pandas has a to_datetime() method for this:

Example

Convert to date:

```
import pandas as pd

df = pd.read_csv('data.csv')

df['Date'] = pd.to_datetime(df['Date'])

print(df.to_string())
```

Result:

```
    Duration          Date  Pulse  Maxpulse  Calories
0         60  '2020/12/01'    110       130     409.1
1         60  '2020/12/02'    117       145     479.0
2         60  '2020/12/03'    103       135     340.0
3         45  '2020/12/04'    109       175     282.4
4         45  '2020/12/05'    117       148     406.0
5         60  '2020/12/06'    102       127     300.0
6         60  '2020/12/07'    110       136     374.0
7        450  '2020/12/08'    104       134     253.3
8         30  '2020/12/09'    109       133     195.1
9         60  '2020/12/10'     98       124     269.0
10        60  '2020/12/11'    103       147     329.3
```

```
11      60  '2020/12/12'   100     120    250.7
12      60  '2020/12/12'   100     120    250.7
13      60  '2020/12/13'   106     128    345.3
14      60  '2020/12/14'   104     132    379.3
15      60  '2020/12/15'    98     123    275.0
16      60  '2020/12/16'    98     120    215.2
17      60  '2020/12/17'   100     120    300.0
18      45  '2020/12/18'    90     112      NaN
19      60  '2020/12/19'   103     123    323.0
20      45  '2020/12/20'    97     125    243.0
21      60  '2020/12/21'   108     131    364.2
22      45          NaT   100     119    282.0
23      60  '2020/12/23'   130     101    300.0
24      45  '2020/12/24'   105     132    246.0
25      60  '2020/12/25'   102     126    334.5
26      60  '2020/12/26'   100     120    250.0
27      60  '2020/12/27'    92     118    241.0
28      60  '2020/12/28'   103     132      NaN
29      60  '2020/12/29'   100     132    280.0
30      60  '2020/12/30'   102     129    380.3
31      60  '2020/12/31'    92     115    243.0
```

As you can see from the result, the date in row 26 was fixed, but the empty date in row 22 got a NaT (Not a Time) value, in other words an empty value. One way to deal with empty values is simply removing the entire row.

Removing Rows

The result from the converting in the example above gave us a NaT value, which can be handled as a NULL value, and we can remove the row by using the dropna() method.

Example

Remove rows with a NULL value in the "Date" column:

```
df.dropna(subset=['Date'], inplace = True)
```

Pandas - Fixing Wrong Data

Wrong Data

"Wrong data" does not have to be "empty cells" or "wrong format", it can just be wrong, like if someone registered "199" instead of "1.99".

Sometimes you can spot wrong data by looking at the data set, because you have an expectation of what it should be.

If you take a look at our data set, you can see that in row 7, the duration is 450, but for all the other rows the duration is between 30 and 60.

It doesn't have to be wrong, but taking in consideration that this is the data set of someone's workout sessions, we conclude with the fact that this person did not work out in 450 minutes.

```
  Duration        Date  Pulse  Maxpulse  Calories
```

```
0        60  '2020/12/01'   110      130    409.1
1        60  '2020/12/02'   117      145    479.0
2        60  '2020/12/03'   103      135    340.0
3        45  '2020/12/04'   109      175    282.4
4        45  '2020/12/05'   117      148    406.0
5        60  '2020/12/06'   102      127    300.0
6        60  '2020/12/07'   110      136    374.0
7       450  '2020/12/08'   104      134    253.3
8        30  '2020/12/09'   109      133    195.1
9        60  '2020/12/10'    98      124    269.0
10       60  '2020/12/11'   103      147    329.3
11       60  '2020/12/12'   100      120    250.7
12       60  '2020/12/12'   100      120    250.7
13       60  '2020/12/13'   106      128    345.3
14       60  '2020/12/14'   104      132    379.3
15       60  '2020/12/15'    98      123    275.0
16       60  '2020/12/16'    98      120    215.2
17       60  '2020/12/17'   100      120    300.0
18       45  '2020/12/18'    90      112      NaN
19       60  '2020/12/19'   103      123    323.0
20       45  '2020/12/20'    97      125    243.0
21       60  '2020/12/21'   108      131    364.2
```

```
22      45          NaN    100      119     282.0
23      60  '2020/12/23'   130      101     300.0
24      45  '2020/12/24'   105      132     246.0
25      60  '2020/12/25'   102      126     334.5
26      60      20201226   100      120     250.0
27      60  '2020/12/27'    92      118     241.0
28      60  '2020/12/28'   103      132       NaN
29      60  '2020/12/29'   100      132     280.0
30      60  '2020/12/30'   102      129     380.3
31      60  '2020/12/31'    92      115     243.0
```

How can we fix wrong values, like the one for "Duration" in row 7?

Replacing Values

One way to fix wrong values is to replace them with something else.

In our example, it is most likely a typo, and the value should be "45" instead of "450", and we could just insert "45" in row 7:

Example

Set "Duration" = 45 in row 7:

```
df.loc[7, 'Duration'] = 45
```

For small data sets you might be able to replace the wrong data one by one, but not for big data sets.

To replace wrong data for larger data sets you can create some rules, e.g. set some boundaries for legal values, and replace any values that are outside of the boundaries.

Example

Loop through all values in the "Duration" column.

If the value is higher than 120, set it to 120:

```
for x in df.index:
  if df.loc[x, "Duration"] > 120:
    df.loc[x, "Duration"] = 120
```

Removing Rows

Another way of handling wrong data is to remove the rows that contains wrong data.

This way you do not have to find out what to replace them with, and there is a good chance you do not need them to do your analyses.

Example

Delete rows where "Duration" is higher than 120:

```
for x in df.index:
  if df.loc[x, "Duration"] > 120:
    df.drop(x, inplace = True)
```

Pandas - Removing Duplicates

Discovering Duplicates

Duplicate rows are rows that have been registered more than one time.

```
  Duration          Date  Pulse  Maxpulse  Calories
0       60  '2020/12/01'    110       130     409.1
1       60  '2020/12/02'    117       145     479.0
2       60  '2020/12/03'    103       135     340.0
3       45  '2020/12/04'    109       175     282.4
4       45  '2020/12/05'    117       148     406.0
5       60  '2020/12/06'    102       127     300.0
6       60  '2020/12/07'    110       136     374.0
7      450  '2020/12/08'    104       134     253.3
8       30  '2020/12/09'    109       133     195.1
9       60  '2020/12/10'     98       124     269.0
10      60  '2020/12/11'    103       147     329.3
11      60  '2020/12/12'    100       120     250.7
12      60  '2020/12/12'    100       120     250.7
13      60  '2020/12/13'    106       128     345.3
14      60  '2020/12/14'    104       132     379.3
15      60  '2020/12/15'     98       123     275.0
```

```
16        60  '2020/12/16'     98       120     215.2
17        60  '2020/12/17'    100       120     300.0
18        45  '2020/12/18'     90       112       NaN
19        60  '2020/12/19'    103       123     323.0
20        45  '2020/12/20'     97       125     243.0
21        60  '2020/12/21'    108       131     364.2
22        45           NaN    100       119     282.0
23        60  '2020/12/23'    130       101     300.0
24        45  '2020/12/24'    105       132     246.0
25        60  '2020/12/25'    102       126     334.5
26        60      20201226    100       120     250.0
27        60  '2020/12/27'     92       118     241.0
28        60  '2020/12/28'    103       132       NaN
29        60  '2020/12/29'    100       132     280.0
30        60  '2020/12/30'    102       129     380.3
31        60  '2020/12/31'     92       115     243.0
```

By taking a look at our test data set, we can assume that row 11 and 12 are duplicates.

To discover duplicates, we can use the duplicated() method.

The duplicated() method returns a Boolean values for each row:

Example

Returns True for every row that is a duplicate, othwerwise False:

```
print(df.duplicated())
```

Removing Duplicates

To remove duplicates, use the drop_duplicates() method.

Example

Remove all duplicates:

```
df.drop_duplicates(inplace = True)
```

Remember: The (inplace = True) will make sure that the method does NOT return a *new* DataFrame, but it will remove all duplicates from the *original* DataFrame.

Chart Visualization

This section demonstrates visualization through charting. For information on visualization of tabular data please see the section on Table Visualization.

We use the standard convention for referencing the matplotlib API:

```
In [1]: import matplotlib.pyplot as plt
```

```
In [2]: plt.close("all")
```

We provide the basics in pandas to easily create decent looking plots. See the ecosystem section for visualization libraries that go beyond the basics documented here.

Note

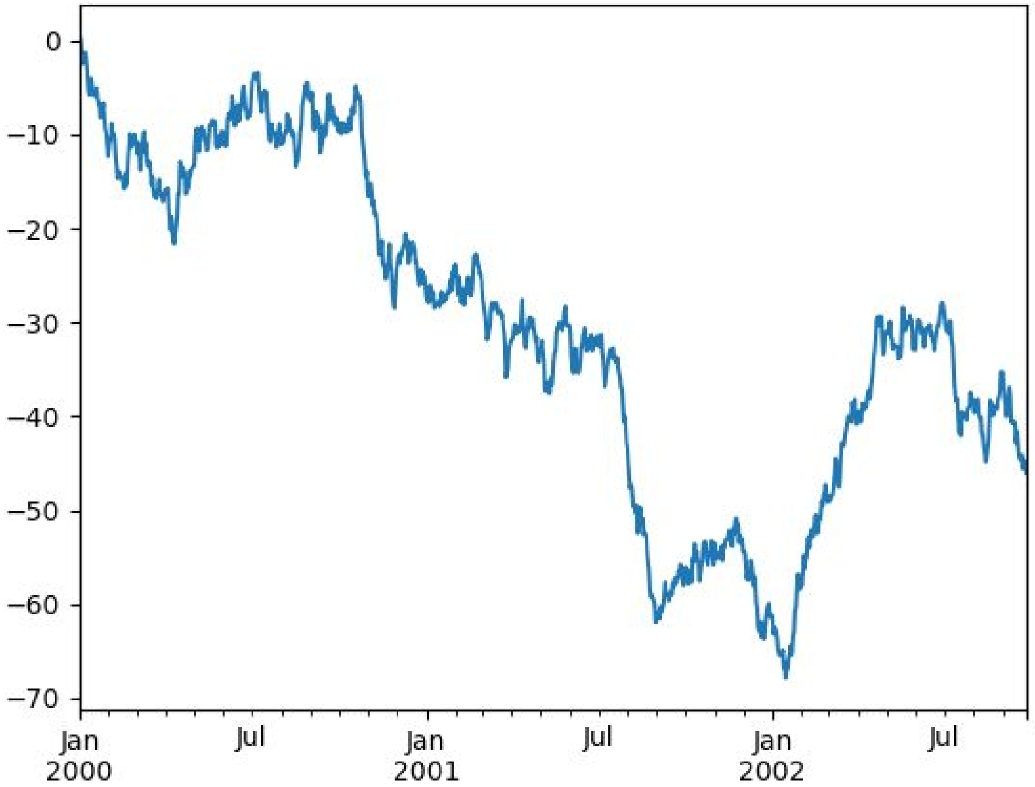
All calls to np.random are seeded with 123456.

Basic plotting: plot

We will demonstrate the basics, see the cookbook for some advanced strategies.

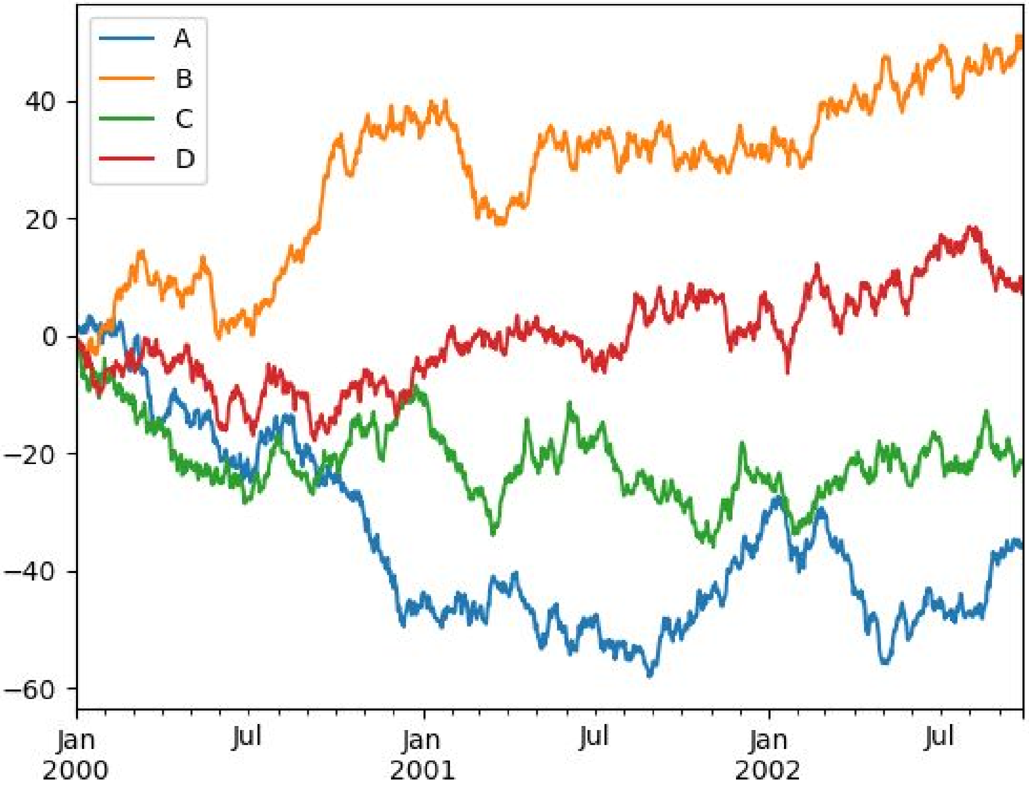
The plot method on Series and DataFrame is just a simple wrapper around plt.plot():

```
In [3]: ts = pd.Series(np.random.randn(1000), index=pd.date_range("1/1/2000", periods=1000))

In [4]: ts = ts.cumsum()

In [5]: ts.plot();
```



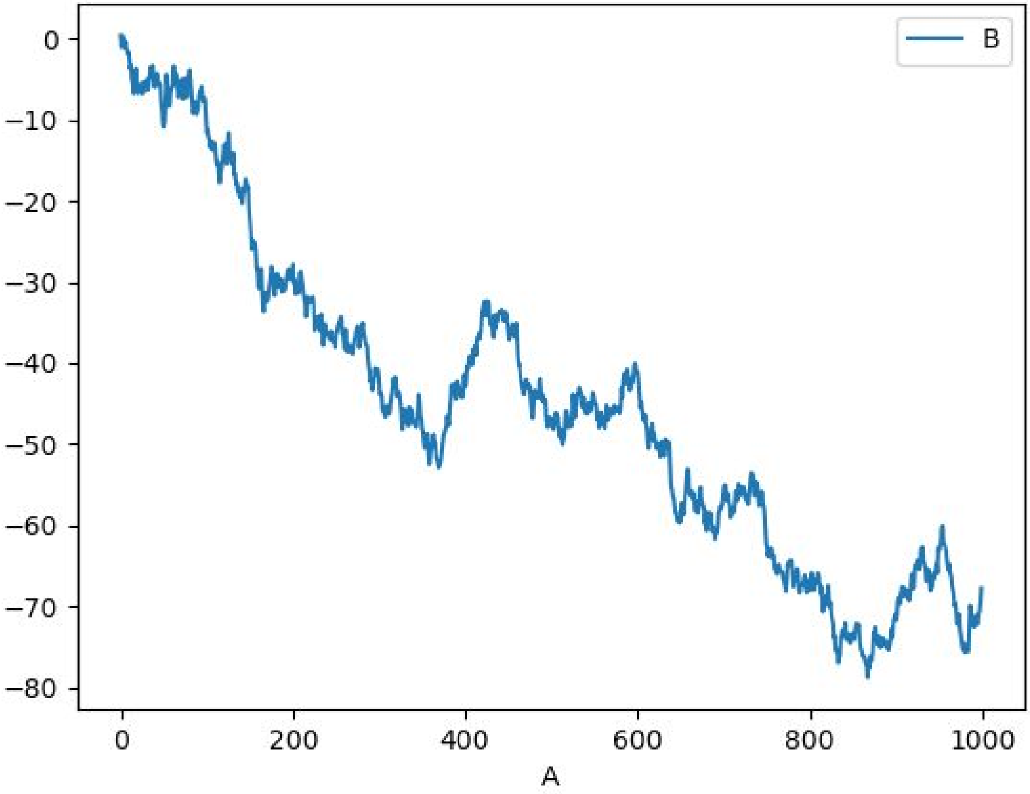
If the index consists of dates, it calls gcf().autofmt_xdate() to try to format the x-axis nicely as per above.

```
In [6]: df = pd.DataFrame(np.random.randn(1000, 4), index=ts.index, columns=list("ABCD"))

In [7]: df = df.cumsum()

In [8]: plt.figure();

In [9]: df.plot();
```



You can plot one column versus another using the x and y keywords in plot():

```
In [10]: df3 = pd.DataFrame(np.random.randn(1000, 2), columns=["B",
"C"]).cumsum()

In [11]: df3["A"] = pd.Series(list(range(len(df))))

In [12]: df3.plot(x="A", y="B");
```



Note

For more formatting and styling options, see formatting below.

Other plots

Plotting methods allow for a handful of plot styles other than the default line plot. These methods can be provided as the kind keyword argument to plot(), and include:

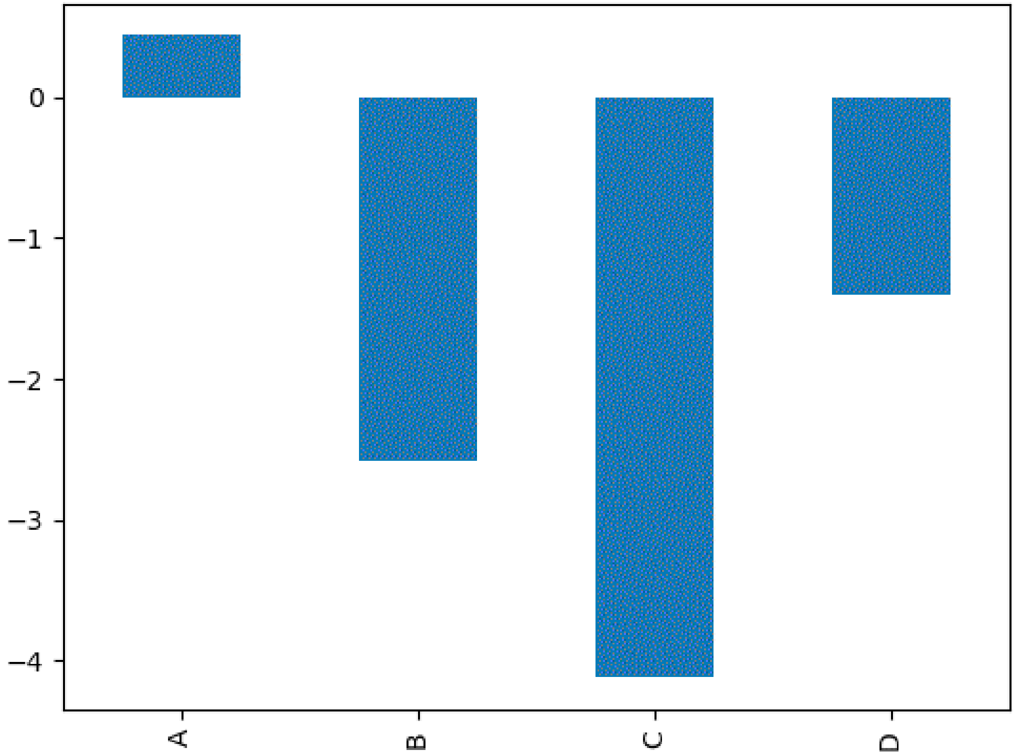
- ‘bar’ or ‘barh’ for bar plots
- ‘hist’ for histogram
- ‘box’ for boxplot
- ‘kde’ or ‘density’ for density plots
- ‘area’ for area plots
- ‘scatter’ for scatter plots

- ‘hexbin’ for hexagonal bin plots
- ‘pie’ for pie plots

For example, a bar plot can be created the following way:

```
In [13]: plt.figure();
```

```
In [14]: df.iloc[5].plot(kind="bar");
```



You can also create these other plots using the methods DataFrame.plot.<kind> instead of providing the kind keyword argument. This makes it easier to discover plot methods and the specific arguments they use:

```
In [15]: df = pd.DataFrame()
```

```
In [16]: df.plot.<TAB>  # noqa: E225, E999
```

df.plot.area df.plot.barh df.plot.density df.plot.hist df.plot.line df.plot.scatter

df.plot.bar df.plot.box df.plot.hexbin df.plot.kde df.plot.pie

In addition to these kind s, there are the DataFrame.hist(), and DataFrame.boxplot() methods, which use a separate interface.

Finally, there are several plotting functions in pandas.plotting that take a Series or DataFrame as an argument. These include:

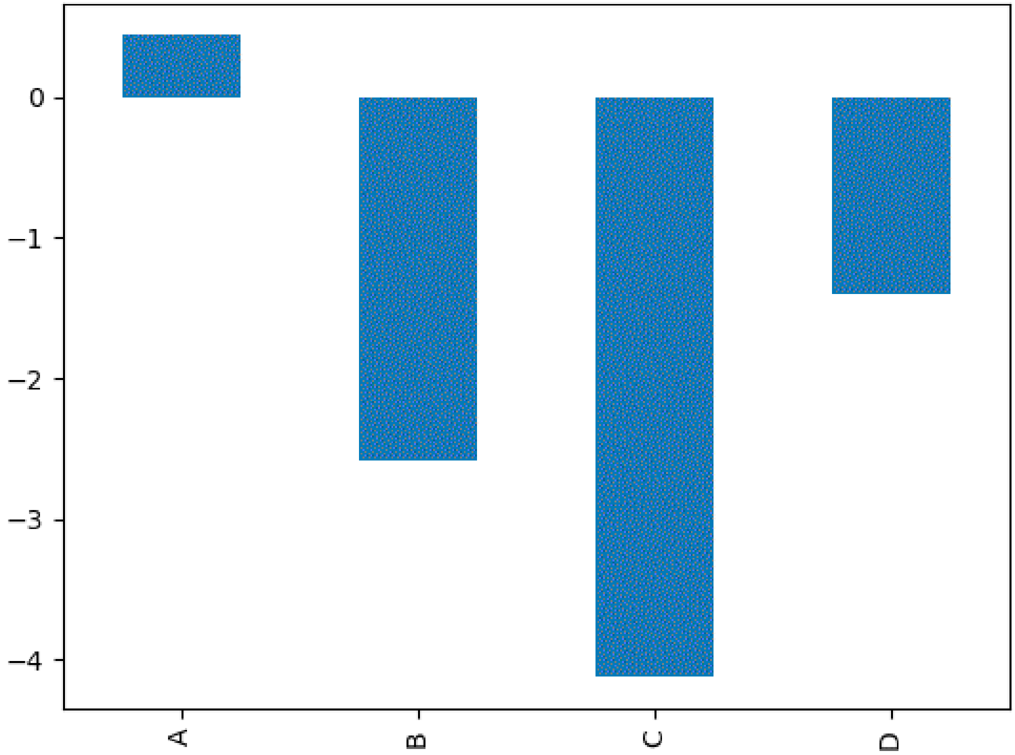
- Scatter Matrix
- Andrews Curves
- Parallel Coordinates
- Lag Plot
- Autocorrelation Plot
- Bootstrap Plot
- RadViz

Plots may also be adorned with errorbars or tables.

Bar plots

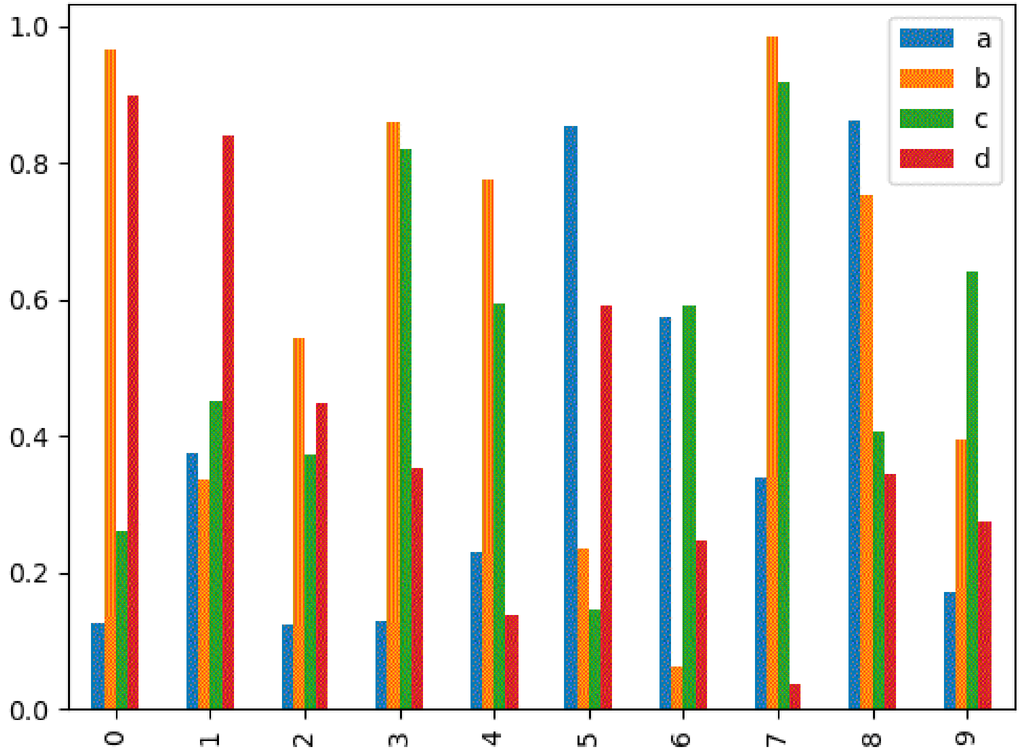
For labeled, non-time series data, you may wish to produce a bar plot:

```
In [17]: plt.figure();

In [18]: df.iloc[5].plot.bar();

In [19]: plt.axhline(0, color="k");
```



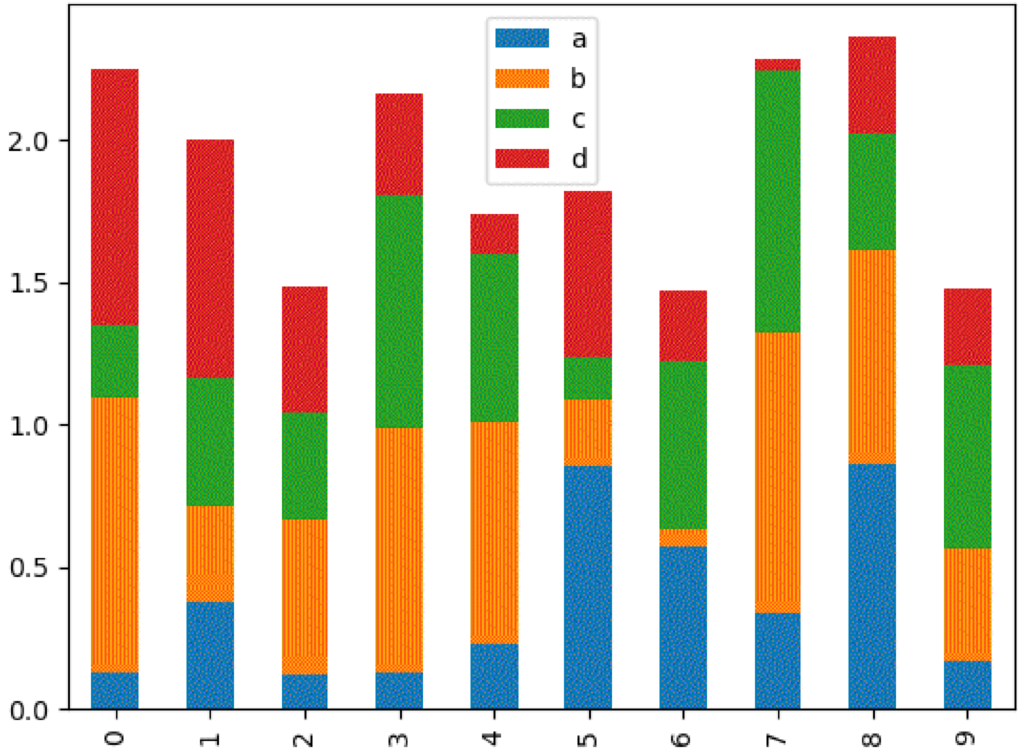
Calling a DataFrame’s plot.bar() method produces a multiple bar plot:

```
In [20]: df2 = pd.DataFrame(np.random.rand(10, 4), columns=["a", "b", "c",
"d"])

In [21]: df2.plot.bar();
```



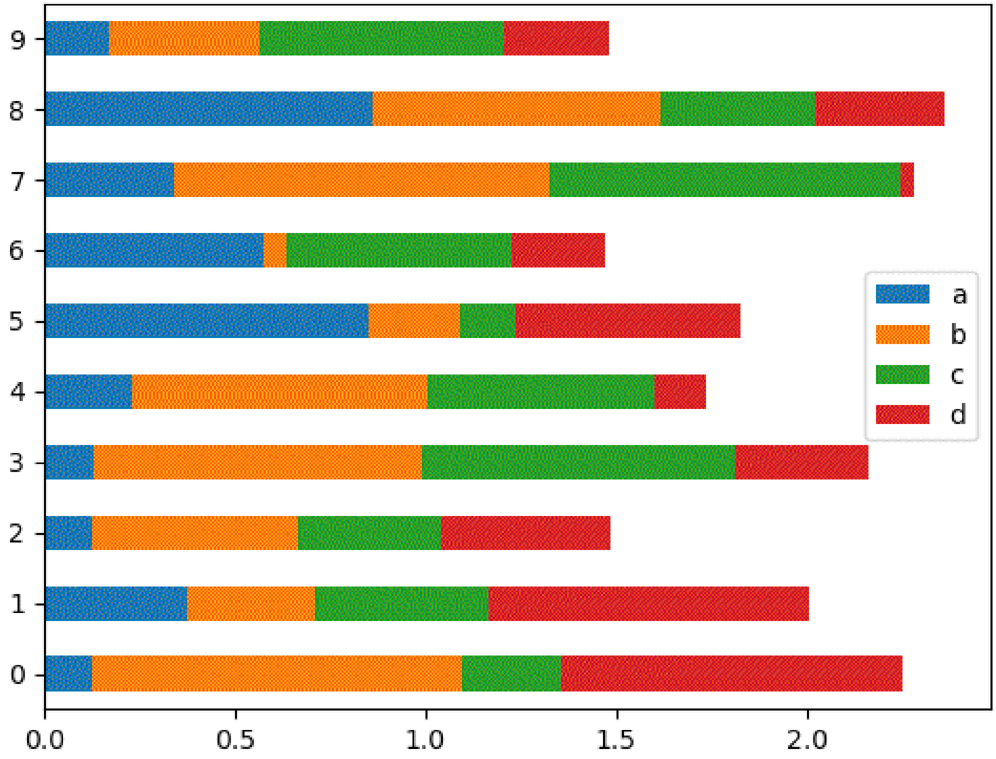
To produce a stacked bar plot, pass stacked=True:

```
In [22]: df2.plot.bar(stacked=True);
```



To get horizontal bar plots, use the barh method:

```
In [23]: df2.plot.barh(stacked=True);
```



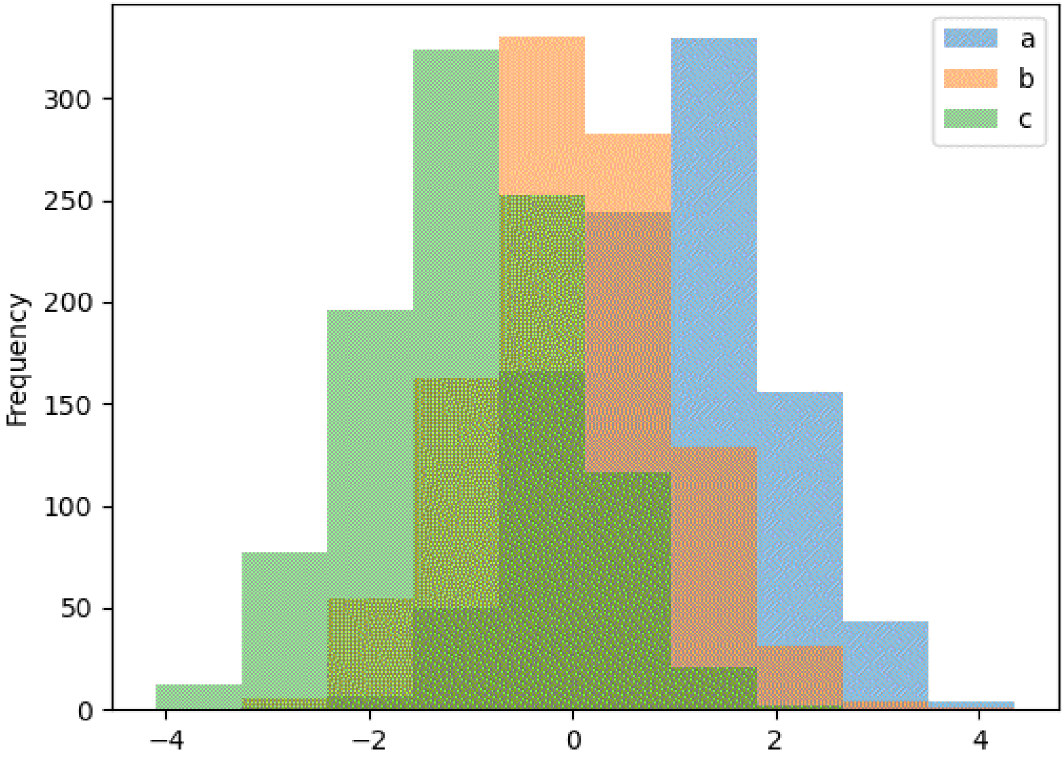
Histograms

Histograms can be drawn by using the DataFrame.plot.hist() and Series.plot.hist() methods.

```
In [24]: df4 = pd.DataFrame(
   ....:     {
   ....:         "a": np.random.randn(1000) + 1,
   ....:         "b": np.random.randn(1000),
   ....:         "c": np.random.randn(1000) - 1,
   ....:     },
   ....:     columns=["a", "b", "c"],
   ....: )

   ....:
```

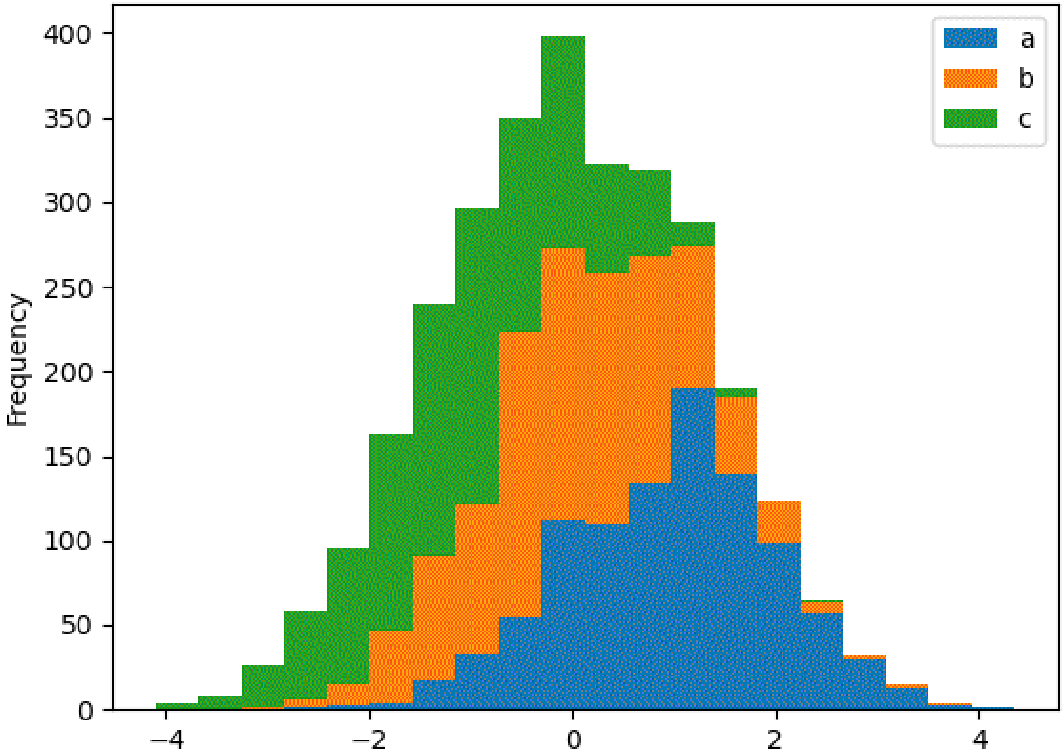
```
In [25]: plt.figure();
```

```
In [26]: df4.plot.hist(alpha=0.5);
```



A histogram can be stacked using stacked=True. Bin size can be changed using the bins keyword.

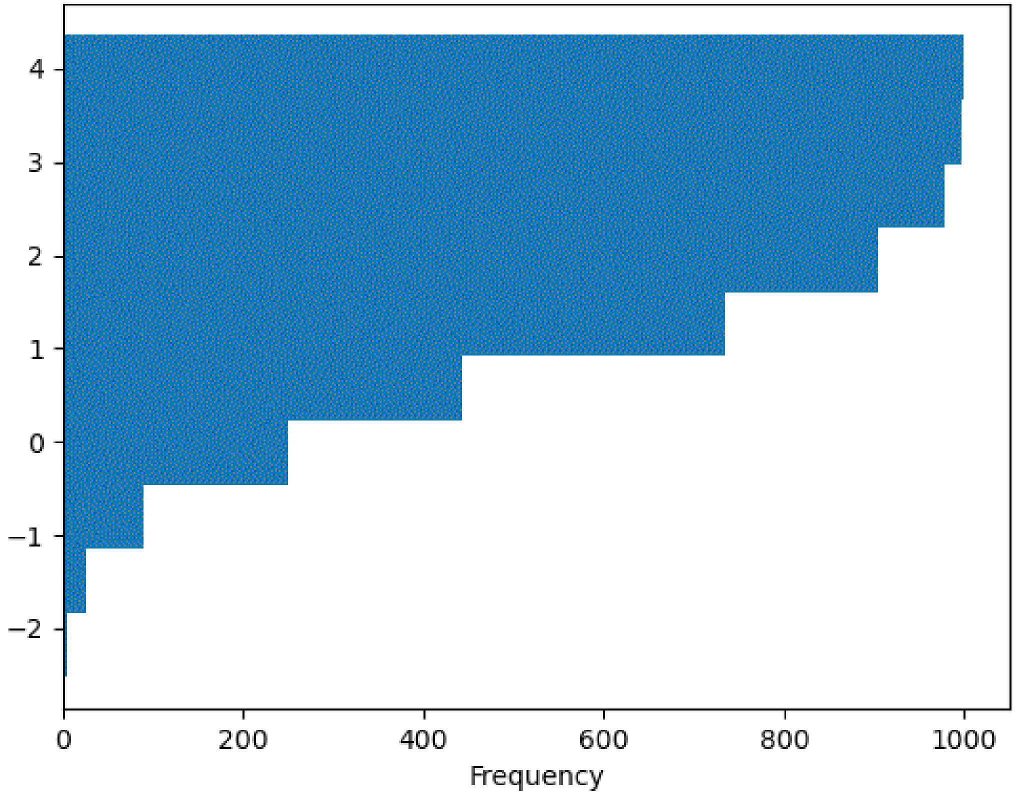
```
In [27]: plt.figure();
In [28]: df4.plot.hist(stacked=True, bins=20);
```



You can pass other keywords supported by matplotlib hist. For example, horizontal and cumulative histograms can be drawn by orientation='horizontal' and cumulative=True.

```
In [29]: plt.figure();
```

```
In [30]: df4["a"].plot.hist(orientation="horizontal", cumulative=True);
```

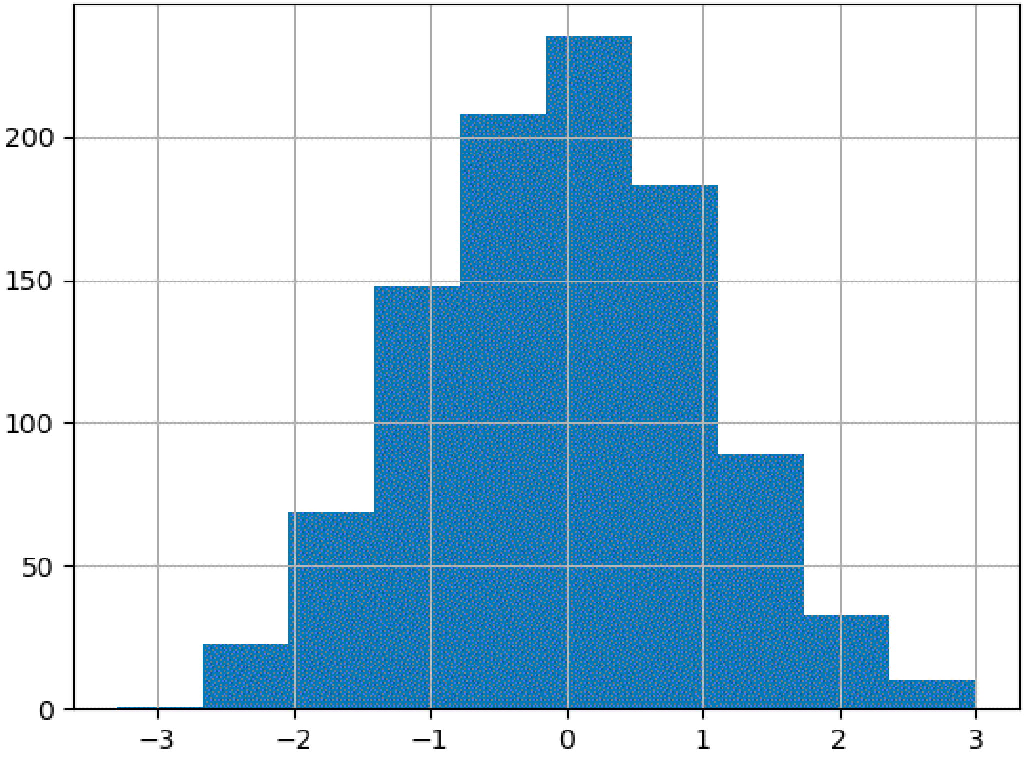


See the hist method and the matplotlib hist documentation for more.

The existing interface DataFrame.hist to plot histogram still can be used.

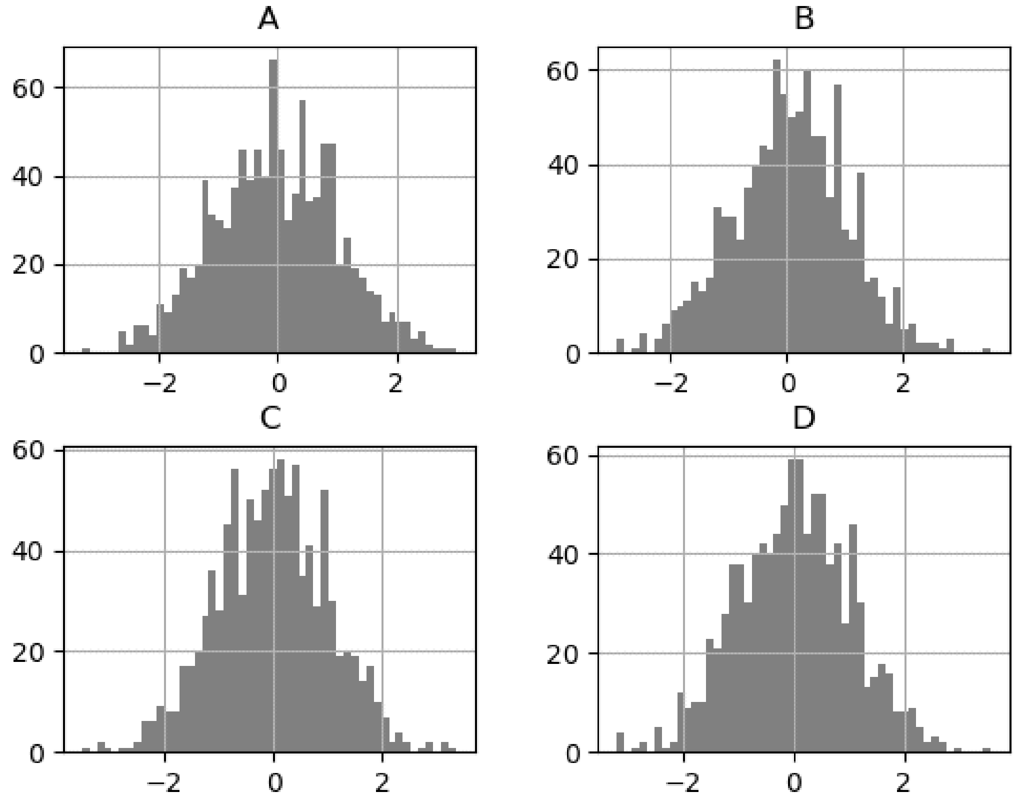
```
In [31]: plt.figure();
```

```
In [32]: df["A"].diff().hist();
```



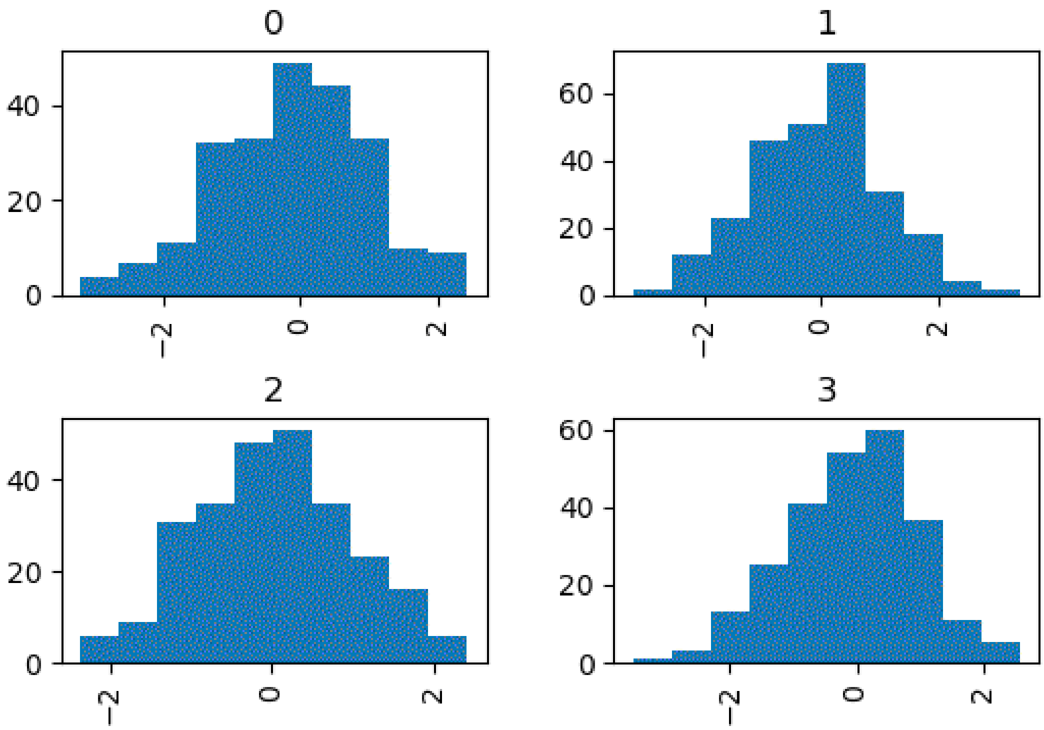
DataFrame.hist() plots the histograms of the columns on multiple subplots:

```
In [33]: plt.figure();

In [34]: df.diff().hist(color="k", alpha=0.5, bins=50);
```



The by keyword can be specified to plot grouped histograms:

```
In [35]: data = pd.Series(np.random.randn(1000))

In [36]: data.hist(by=np.random.randint(0, 4, 1000), figsize=(6, 4));
```



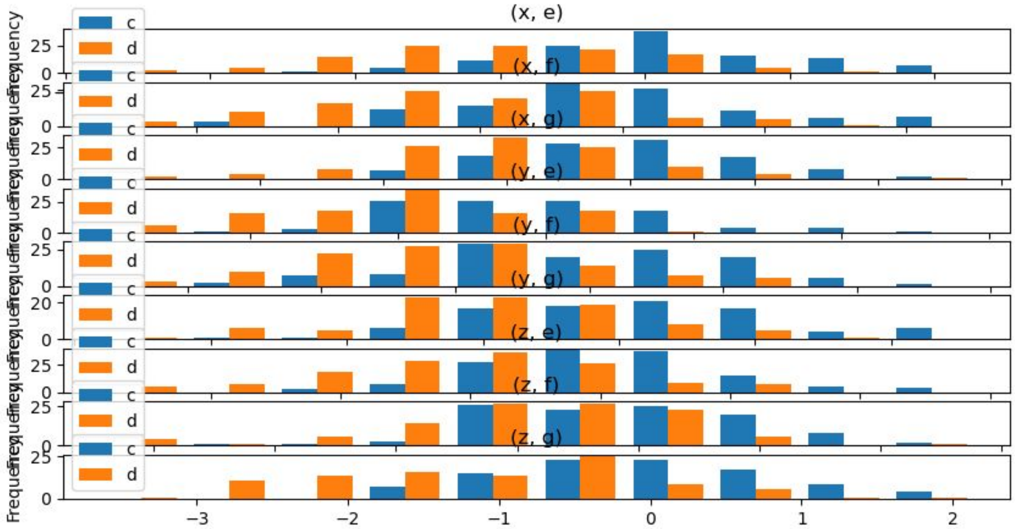
In addition, the by keyword can also be specified in DataFrame.plot.hist().

Changed in version 1.4.0.

```
In [37]: data = pd.DataFrame(
   ....:     {
   ....:         "a": np.random.choice(["x", "y", "z"], 1000),
   ....:         "b": np.random.choice(["e", "f", "g"], 1000),
   ....:         "c": np.random.randn(1000),
   ....:         "d": np.random.randn(1000) - 1,
   ....:     },
   ....: )
```

```
....:
```

```
In [38]: data.plot.hist(by=["a", "b"], figsize=(10, 5));
```



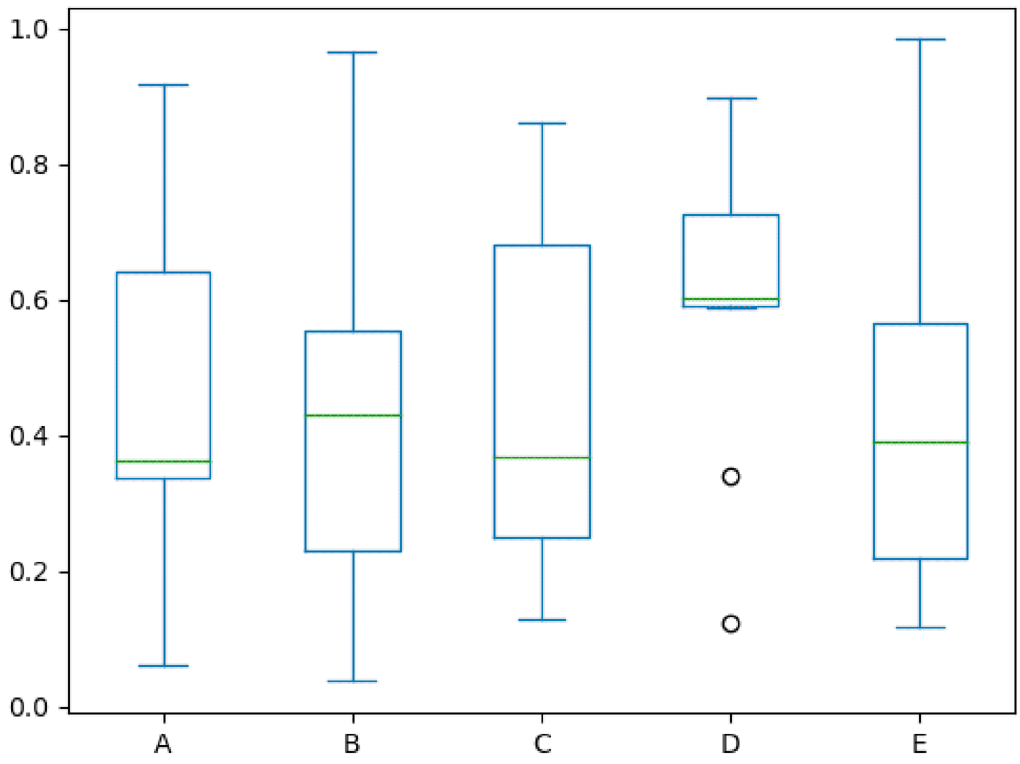
Box plots

Boxplot can be drawn calling Series.plot.box() and DataFrame.plot.box(), or DataFrame.boxplot() to visualize the distribution of values within each column.

For instance, here is a boxplot representing five trials of 10 observations of a uniform random variable on [0,1).

```
In [39]: df = pd.DataFrame(np.random.rand(10, 5), columns=["A", "B", "C",
"D", "E"])
```

```
In [40]: df.plot.box();
```



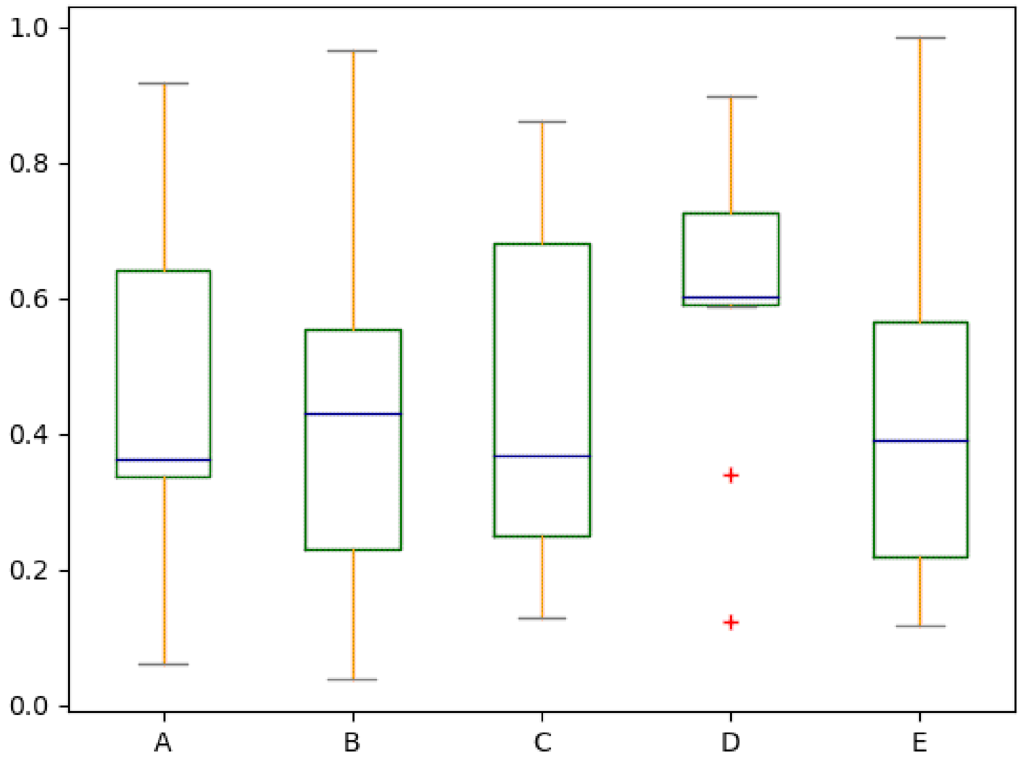
Boxplot can be colorized by passing color keyword. You can pass a dict whose keys are boxes, whiskers, medians and caps. If some keys are missing in the dict, default colors are used for the corresponding artists. Also, boxplot has sym keyword to specify fliers style.

When you pass other type of arguments via color keyword, it will be directly passed to matplotlib for all the boxes, whiskers, medians and caps colorization.

The colors are applied to every boxes to be drawn. If you want more complicated colorization, you can get each drawn artists by passing return_type.

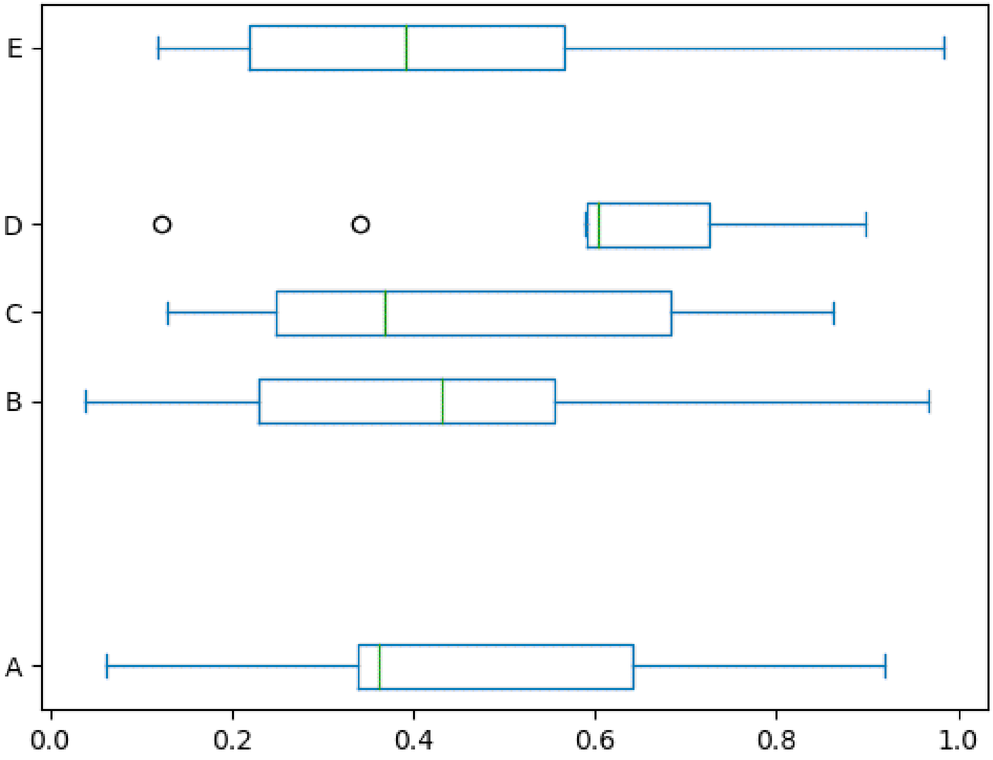
```
In [41]: color = {

 ....:     "boxes": "DarkGreen",

 ....:     "whiskers": "DarkOrange",
 ....:     "medians": "DarkBlue",
 ....:     "caps": "Gray",
 ....: }
 ....:
```

```
In [42]: df.plot.box(color=color, sym="r+");
```



Also, you can pass other keywords supported by matplotlib boxplot. For example, horizontal and custom-positioned boxplot can be drawn by vert=False and positions keywords.

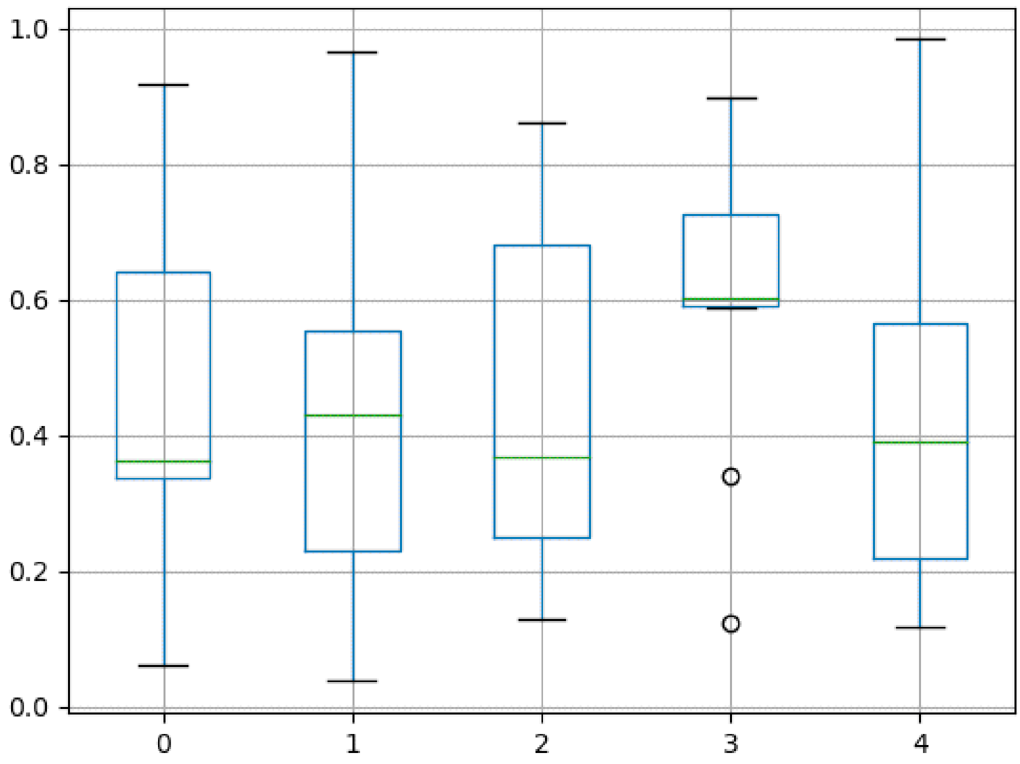
```
In [43]: df.plot.box(vert=False, positions=[1, 4, 5, 6, 8]);
```



See the boxplot method and the matplotlib boxplot documentation for more.

The existing interface DataFrame.boxplot to plot boxplot still can be used.

```
In [44]: df = pd.DataFrame(np.random.rand(10, 5))

In [45]: plt.figure();

In [46]: bp = df.boxplot()
```



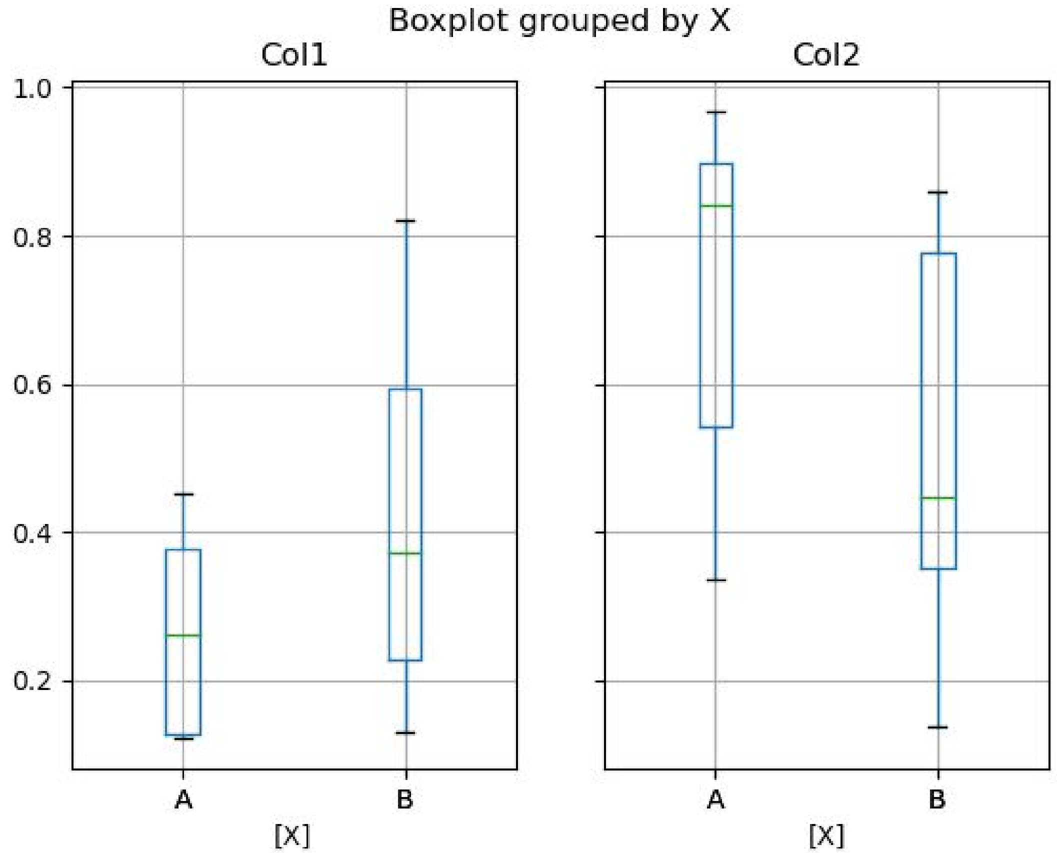
You can create a stratified boxplot using the by keyword argument to create groupings. For instance,

```
In [47]: df = pd.DataFrame(np.random.rand(10, 2), columns=["Col1",
"Col2"])
```

```
In [48]: df["X"] = pd.Series(["A", "A", "A", "A", "A", "B", "B", "B", "B",
"B"])
```

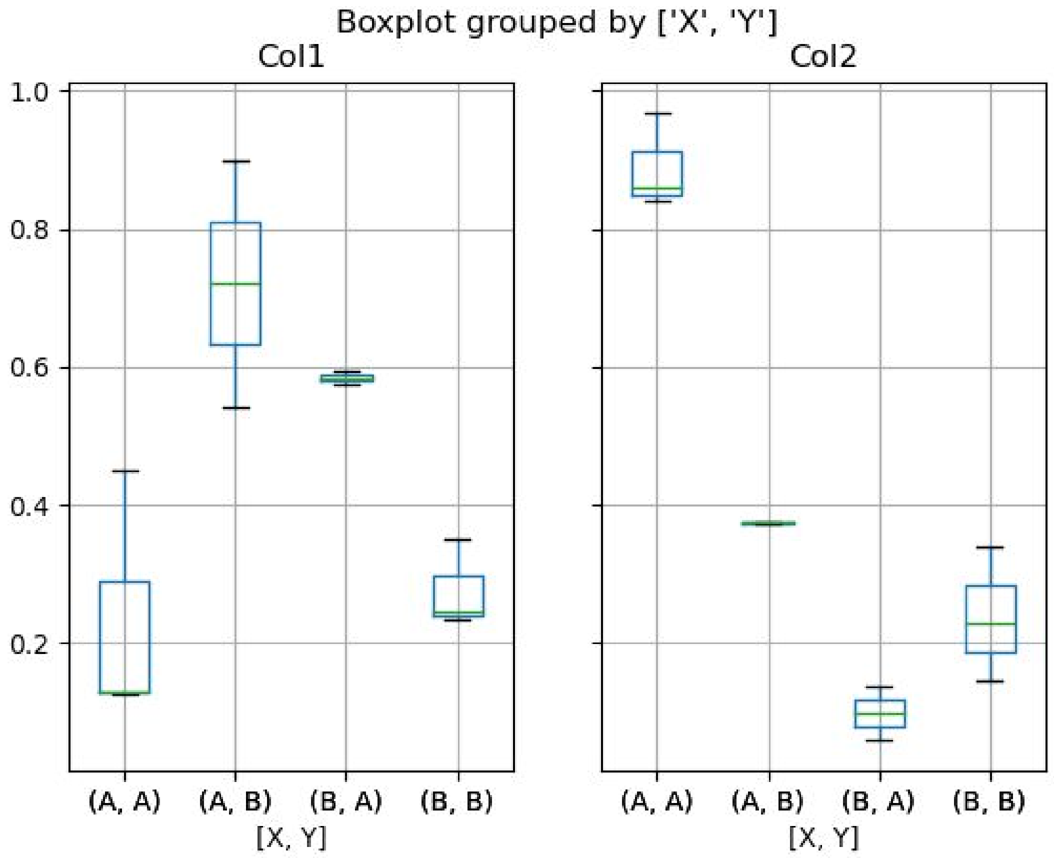
```
In [49]: plt.figure();
```

```
In [50]: bp = df.boxplot(by="X")
```



You can also pass a subset of columns to plot, as well as group by multiple columns:

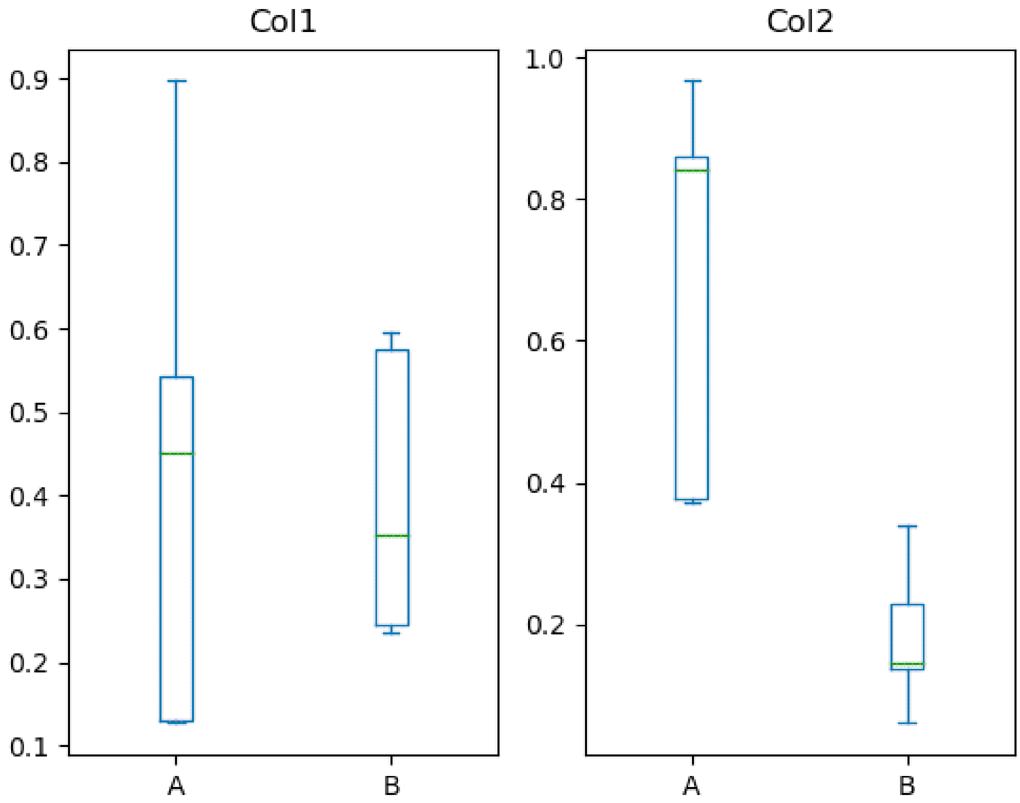
```
In [51]: df = pd.DataFrame(np.random.rand(10, 3), columns=["Col1", "Col2",
"Col3"])

In [52]: df["X"] = pd.Series(["A", "A", "A", "A", "A", "B", "B", "B", "B",
"B"])


In [53]: df["Y"] = pd.Series(["A", "B", "A", "B", "A", "B", "A", "B", "A",
"B"])

In [54]: plt.figure();

In [55]: bp = df.boxplot(column=["Col1", "Col2"], by=["X", "Y"])
```



You could also create groupings with DataFrame.plot.box(), for instance:

Changed in version 1.4.0.

```
In [56]: df = pd.DataFrame(np.random.rand(10, 3), columns=["Col1", "Col2", "Col3"])

In [57]: df["X"] = pd.Series(["A", "A", "A", "A", "A", "B", "B", "B", "B", "B"])

In [58]: plt.figure();

In [59]: bp = df.plot.box(column=["Col1", "Col2"], by="X")
```



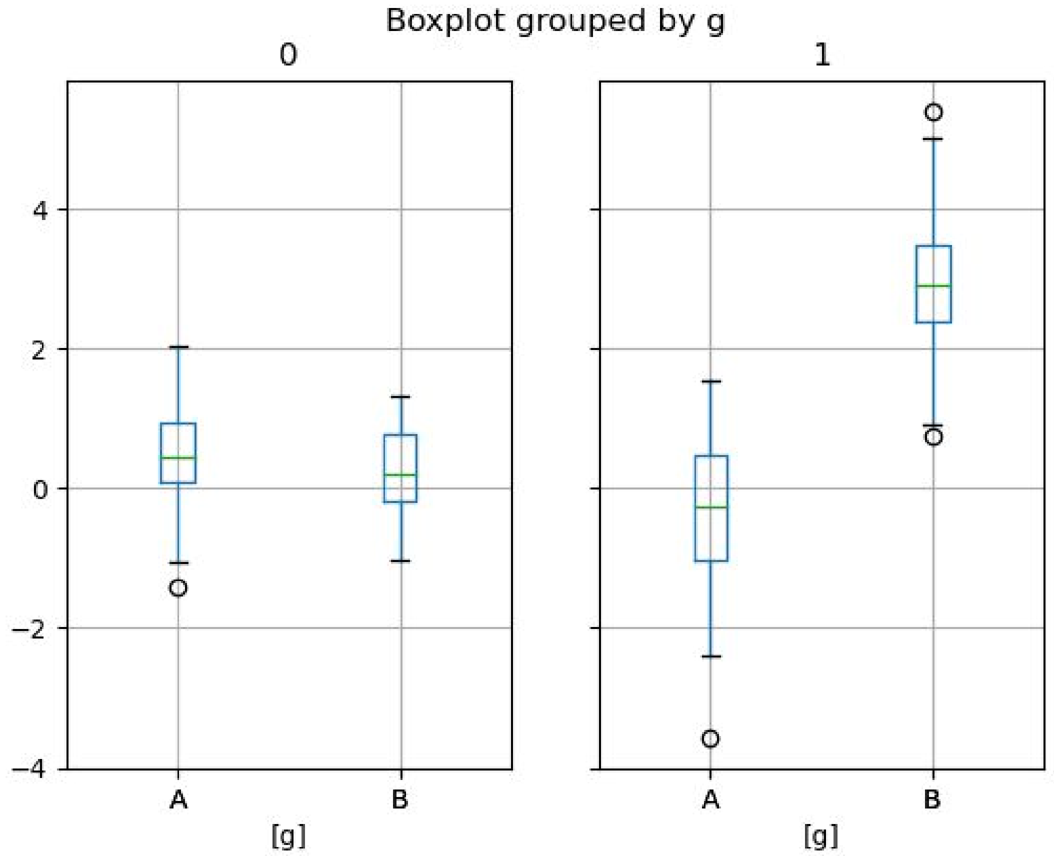
In boxplot, the return type can be controlled by the return_type, keyword. The valid choices are {"axes", "dict", "both", None}. Faceting, created by DataFrame.boxplot with the by keyword, will affect the output type as well:

return_type	Faceted	Output type
None	No	axes
None	Yes	2-D ndarray of axes
'axes'	No	axes
'axes'	Yes	Series of axes

'dict'	No	dict of artists
'dict'	Yes	Series of dicts of artists
'both'	No	namedtuple
'both'	Yes	Series of namedtuples

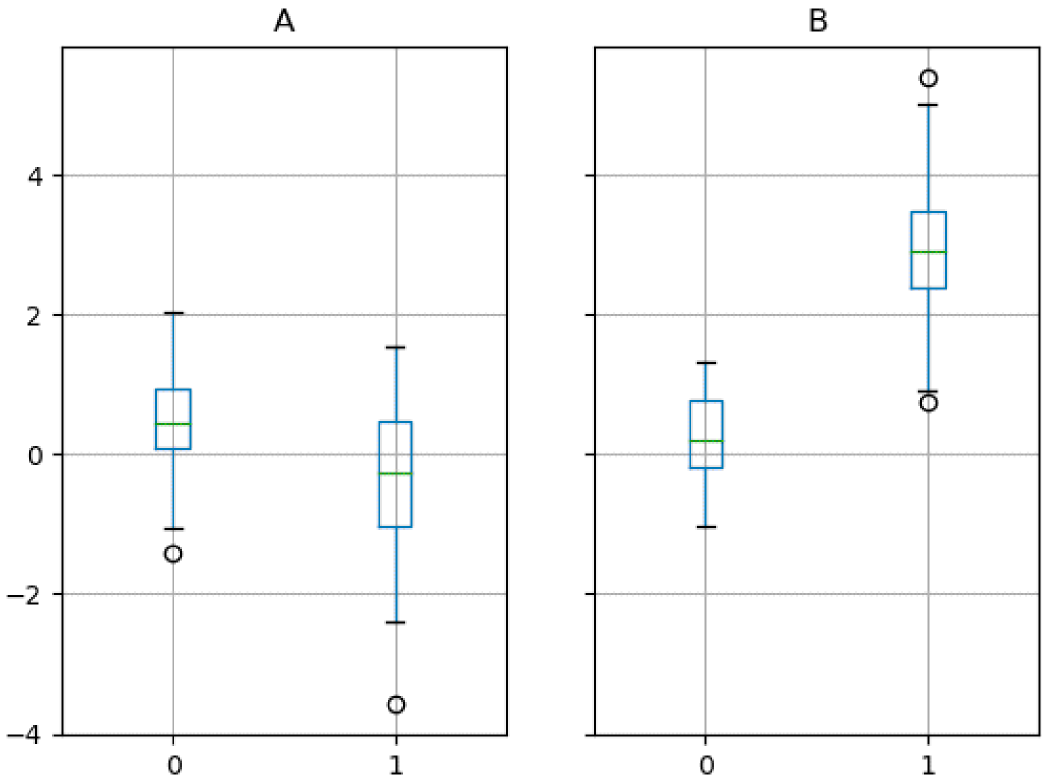
Groupby.boxplot always returns a Series of return_type.

```
In [60]: np.random.seed(1234)

In [61]: df_box = pd.DataFrame(np.random.randn(50, 2))


In [62]: df_box["g"] = np.random.choice(["A", "B"], size=50

In [63]: df_box.loc[df_box["g"] == "B", 1] += 3

In [64]: bp = df_box.boxplot(by="g")
```



The subplots above are split by the numeric columns first, then the value of the g column. Below the subplots are first split by the value of g, then by the numeric columns.

```
In [65]: bp = df_box.groupby("g").boxplot()
```



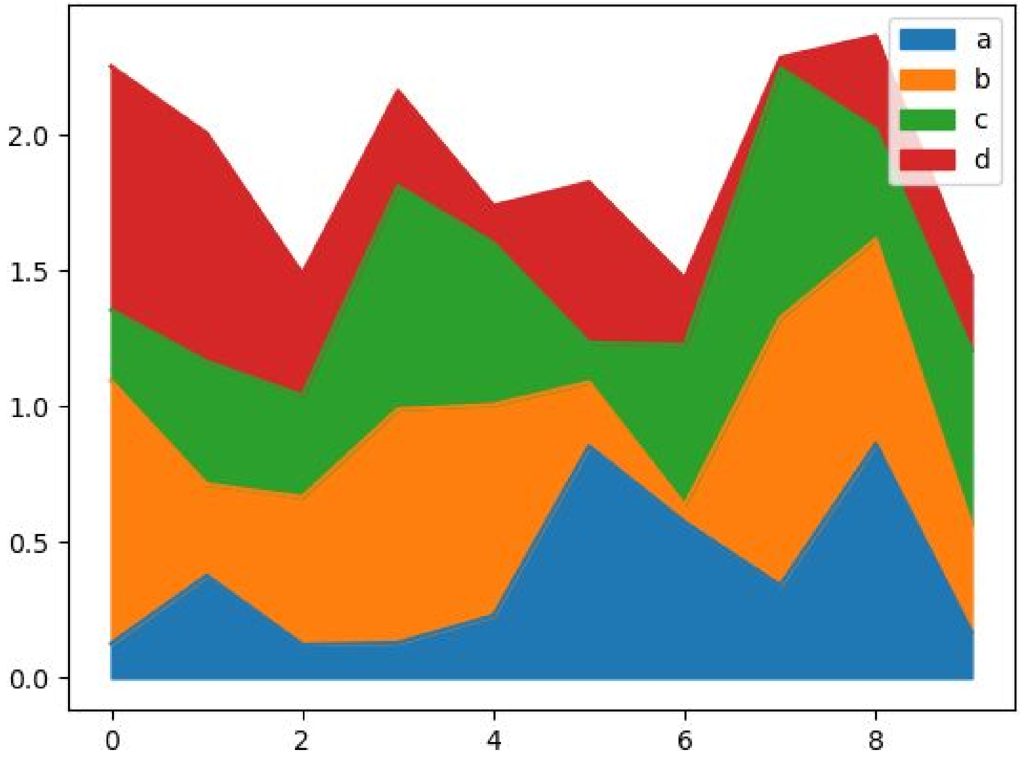
Area plot

You can create area plots with Series.plot.area() and DataFrame.plot.area(). Area plots are stacked by default. To produce stacked area plot, each column must be either all positive or all negative values.

When input data contains NaN, it will be automatically filled by 0. If you want to drop or fill by different values, use dataframe.dropna() or dataframe.fillna() before calling plot.

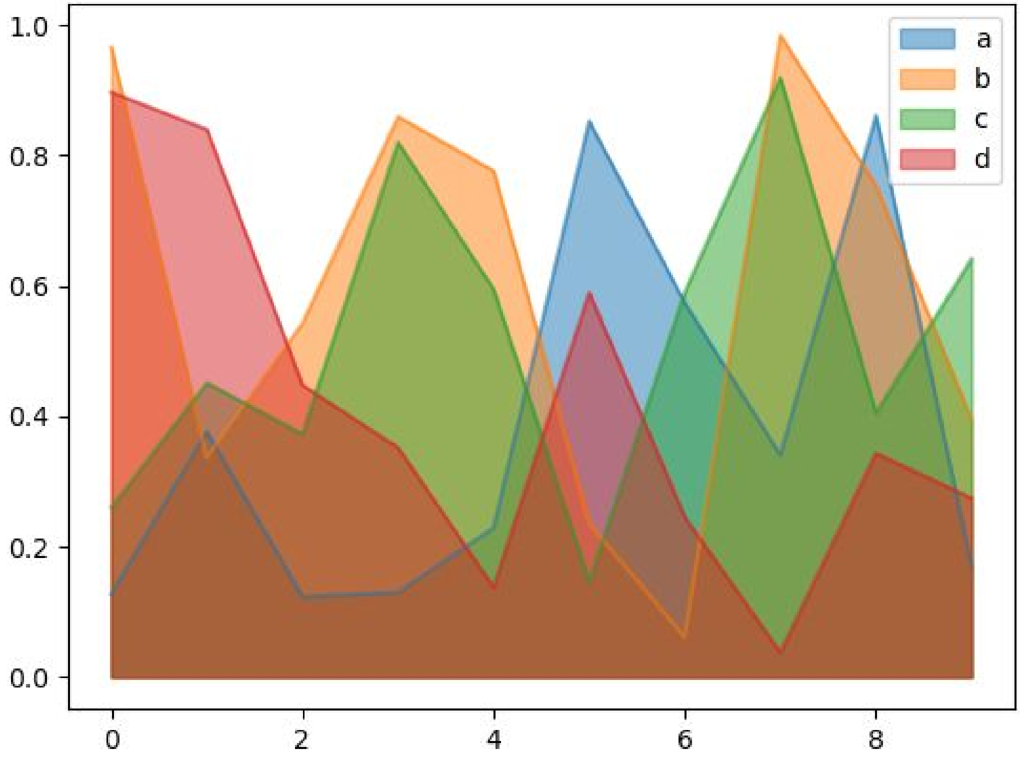
```
In [66]: df = pd.DataFrame(np.random.rand(10, 4), columns=["a", "b", "c",
"d"])
```

```
In [67]: df.plot.area();
```



To produce an unstacked plot, pass stacked=False. Alpha value is set to 0.5 unless otherwise specified:

```
In [68]: df.plot.area(stacked=False);
```



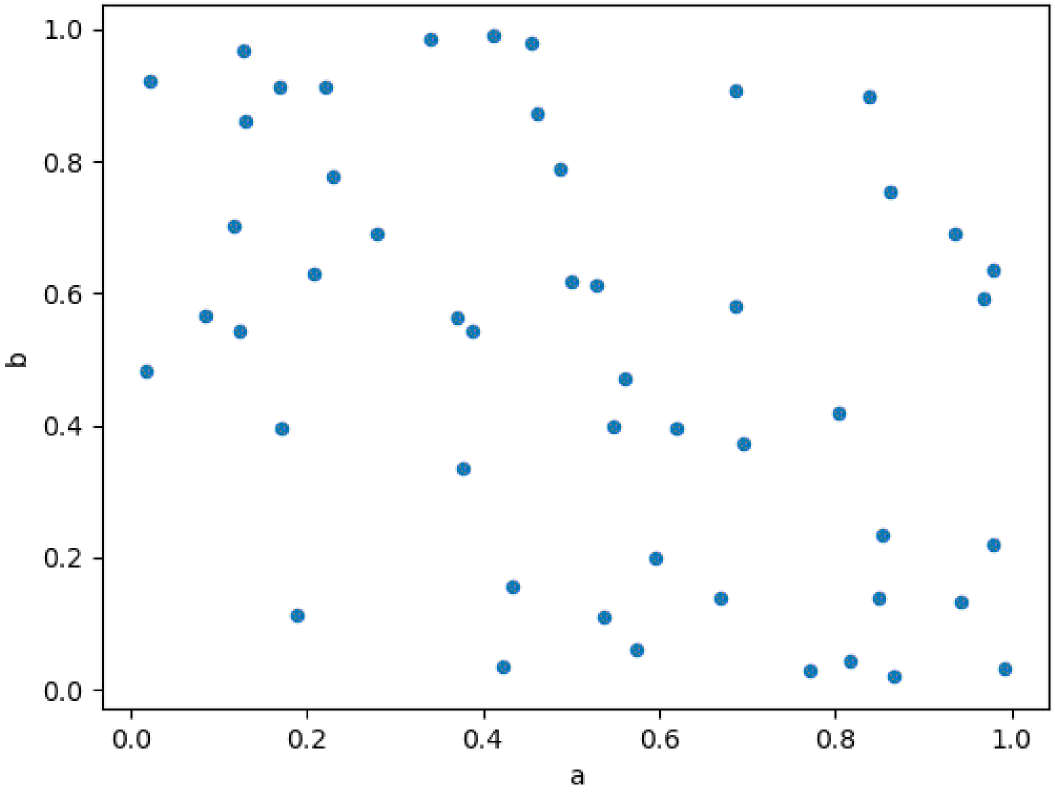
Scatter plot

Scatter plot can be drawn by using the DataFrame.plot.scatter() method. Scatter plot requires numeric columns for the x and y axes. These can be specified by the x and y keywords.

```
In [69]: df = pd.DataFrame(np.random.rand(50, 4), columns=["a", "b", "c",
"d"])


In [70]: df["species"] = pd.Categorical(

   ....:     ["setosa"] * 20 + ["versicolor"] * 20 + ["virginica"] * 10

   ....: )
```

```
....:
```

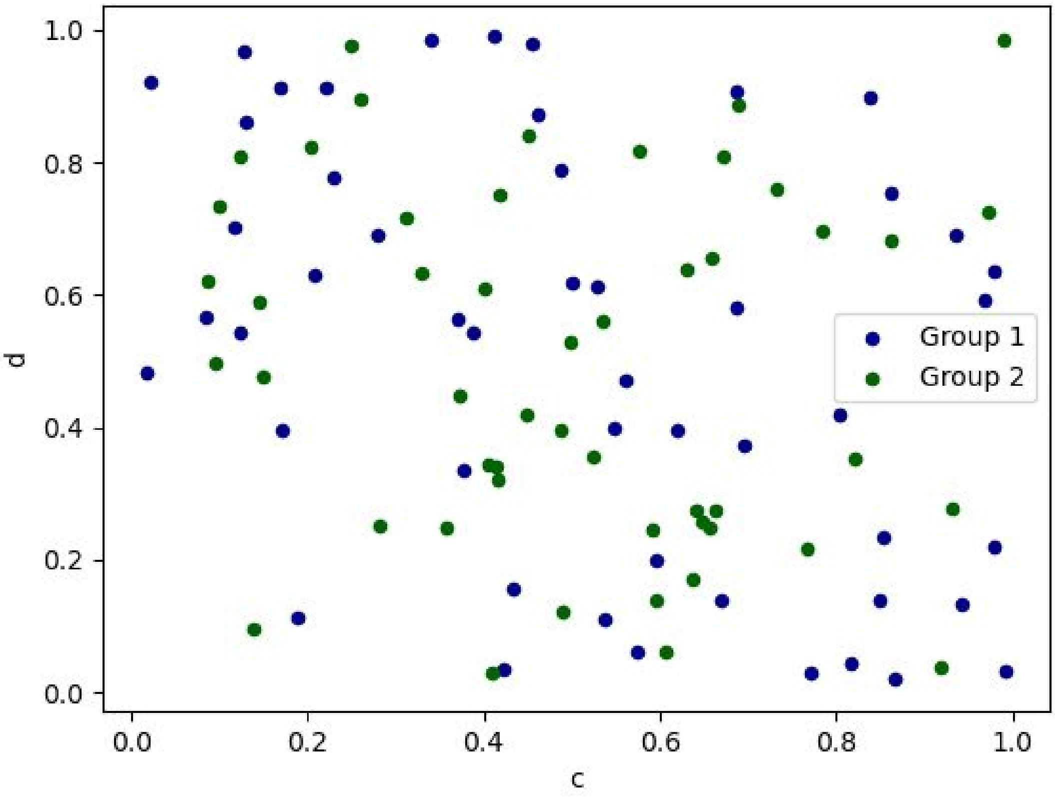
```
In [71]: df.plot.scatter(x="a", y="b");
```



To plot multiple column groups in a single axes, repeat plot method specifying target ax. It is recommended to specify color and label keywords to distinguish each groups.

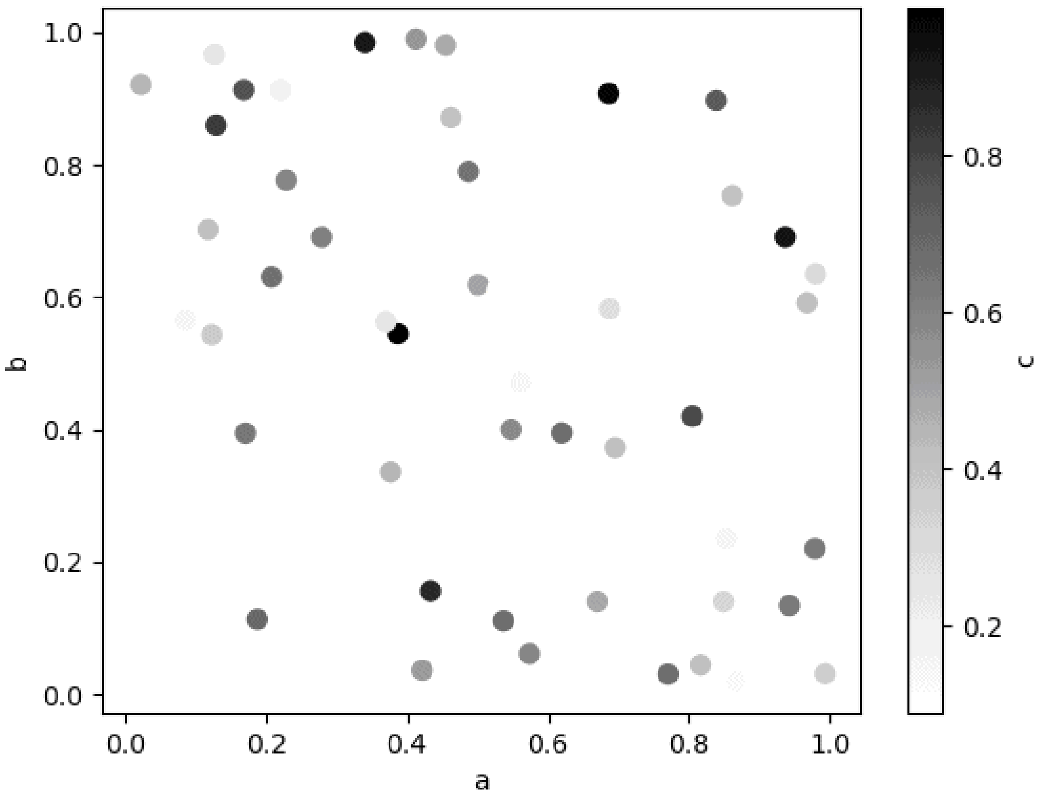
```
In [72]: ax = df.plot.scatter(x="a", y="b", color="DarkBlue", label="Group
1")
```

```
In [73]: df.plot.scatter(x="c", y="d", color="DarkGreen", label="Group 2",
ax=ax);
```



The keyword c may be given as the name of a column to provide colors for each point:

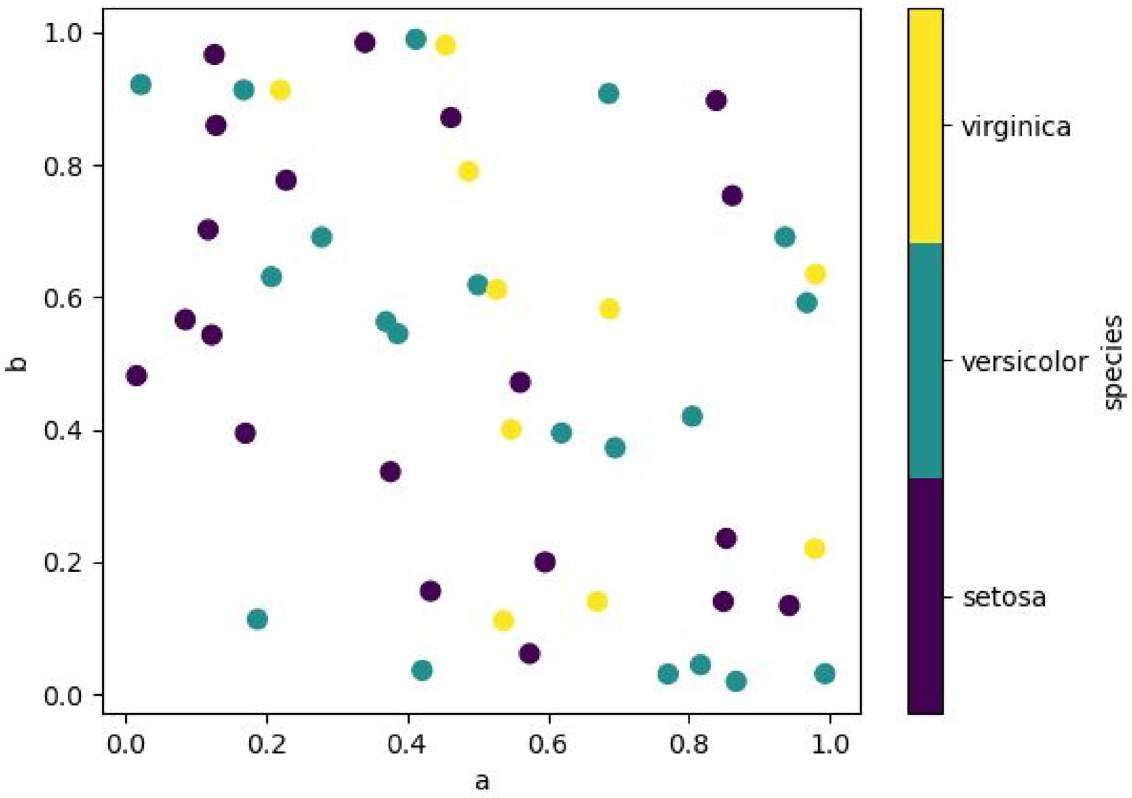
```
In [74]: df.plot.scatter(x="a", y="b", c="c", s=50);
```



If a categorical column is passed to c, then a discrete colorbar will be produced:

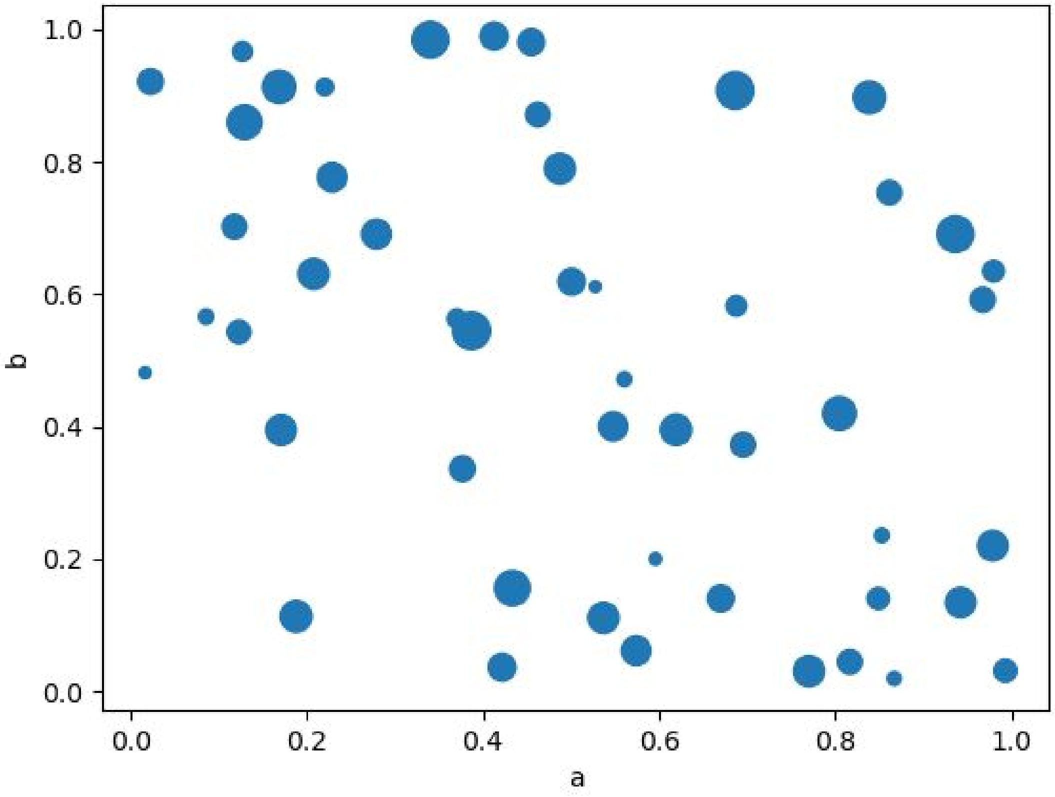
New in version 1.3.0.

```
In [75]: df.plot.scatter(x="a", y="b", c="species", cmap="viridis", s=50);
```



You can pass other keywords supported by matplotlib scatter. The example below shows a bubble chart using a column of the DataFrame as the bubble size.

```
In [76]: df.plot.scatter(x="a", y="b", s=df["c"] * 200);
```



See the scatter method and the matplotlib scatter documentation for more.

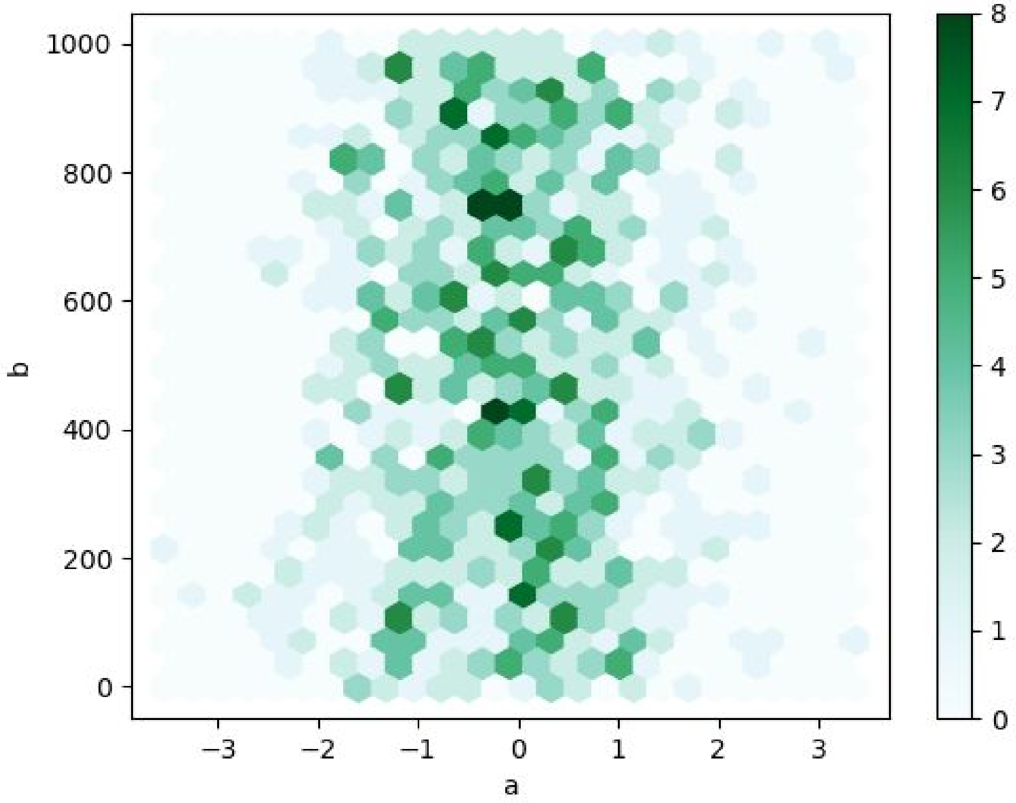
Hexagonal bin plot

You can create hexagonal bin plots with DataFrame.plot.hexbin(). Hexbin plots can be a useful alternative to scatter plots if your data are too dense to plot each point individually.

```
In [77]: df = pd.DataFrame(np.random.randn(1000, 2), columns=["a", "b"])
```

```
In [78]: df["b"] = df["b"] + np.arange(1000)
```

```
In [79]: df.plot.hexbin(x="a", y="b", gridsize=25);
```



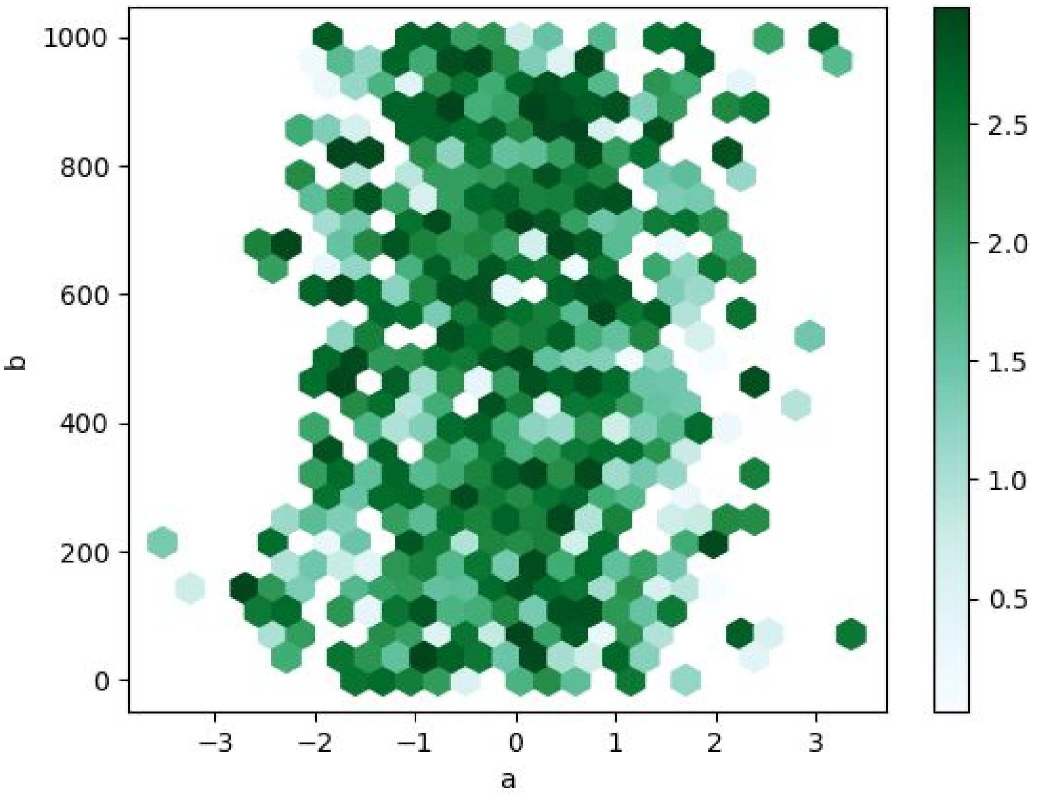
A useful keyword argument is gridsize; it controls the number of hexagons in the x-direction, and defaults to 100. A larger gridsize means more, smaller bins.

By default, a histogram of the counts around each (x, y) point is computed. You can specify alternative aggregations by passing values to the C and reduce_C_function arguments. C specifies the value at each (x, y) point and reduce_C_function is a function of one argument that reduces all the values in a bin to a single number (e.g. mean, max, sum, std). In this example the positions are given by columns a and b, while the value is given by column z. The bins are aggregated with NumPy's max function.

```
In [80]: df = pd.DataFrame(np.random.randn(1000, 2), columns=["a", "b"])
```

```
In [81]: df["b"] = df["b"] + np.arange(1000)
In [82]: df["z"] = np.random.uniform(0, 3, 1000)
```

```
In [83]: df.plot.hexbin(x="a", y="b", C="z", reduce_C_function=np.max, gridsize=25);
```

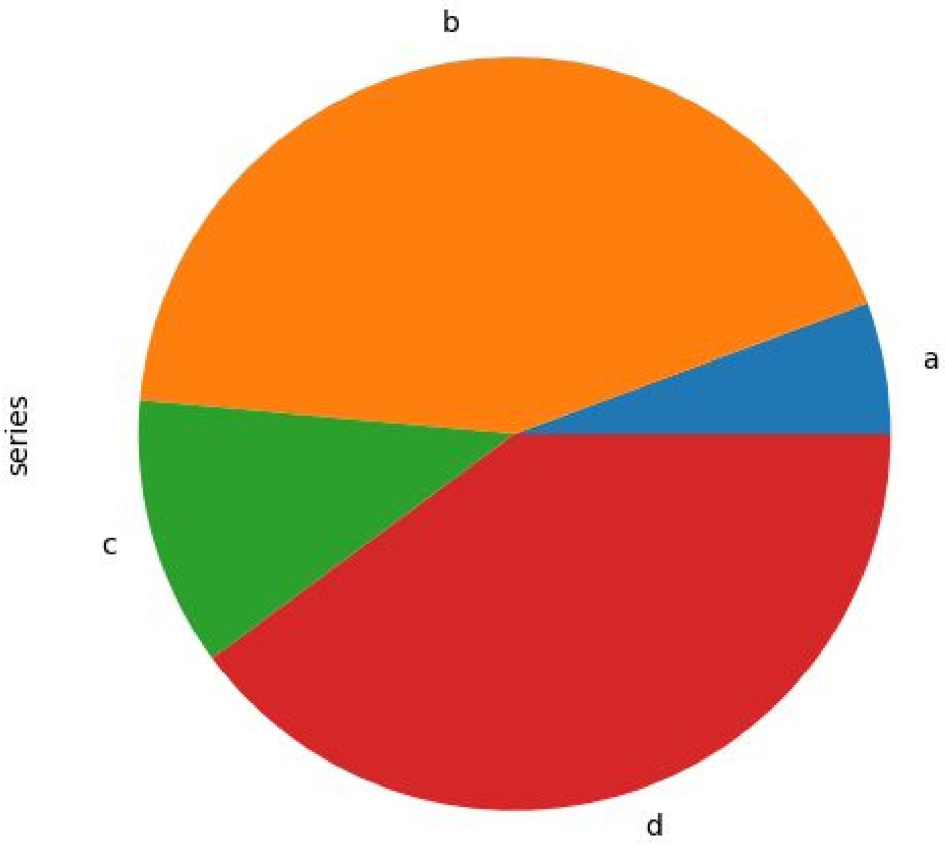


See the hexbin method and the matplotlib hexbin documentation for more.

Pie plot

You can create a pie plot with DataFrame.plot.pie() or Series.plot.pie(). If your data includes any NaN, they will be automatically filled with 0. A ValueError will be raised if there are any negative values in your data.

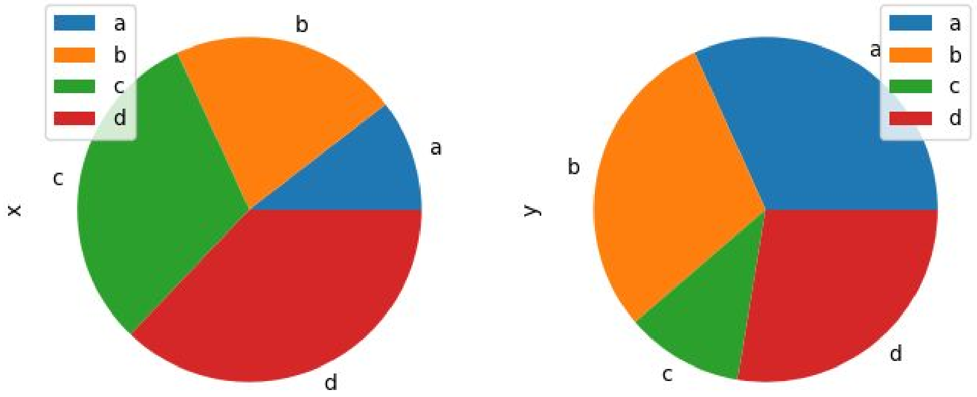
```
In [84]: series = pd.Series(3 * np.random.rand(4), index=["a", "b", "c", "d"], name="series")
In [85]: series.plot.pie(figsize=(6, 6));
```



For pie plots it’s best to use square figures, i.e. a figure aspect ratio 1. You can create the figure with equal width and height, or force the aspect ratio to be equal after plotting by calling ax.set_aspect('equal') on the returned axes object.

Note that pie plot with DataFrame requires that you either specify a target column by the y argument or subplots=True. When y is specified, pie plot of selected column will be drawn. If subplots=True is specified, pie plots for each column are drawn as subplots. A legend will be drawn in each pie plots by default; specify legend=False to hide it.

```
In [86]: df = pd.DataFrame(
   ....:     3 * np.random.rand(4, 2), index=["a", "b", "c", "d"], columns=["x", "y"]
   ....: )
   ....:

In [87]: df.plot.pie(subplots=True, figsize=(8, 4));
```



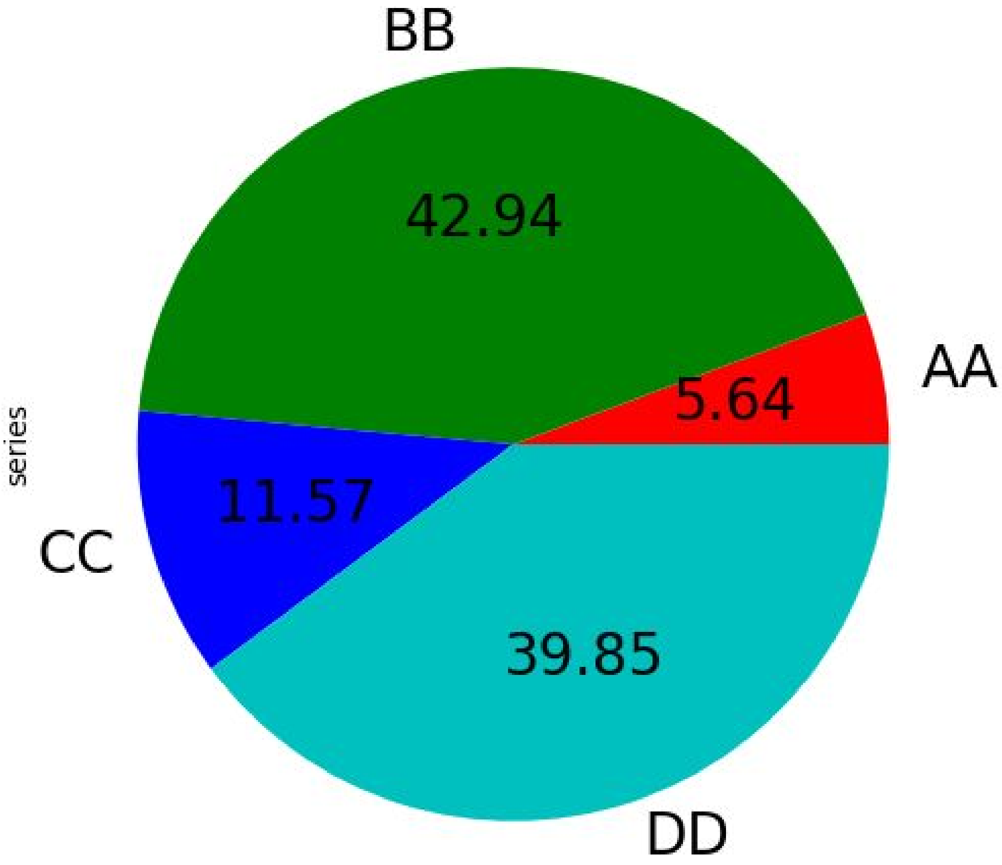
You can use the labels and colors keywords to specify the labels and colors of each wedge.

Warning

Most pandas plots use the label and color arguments (note the lack of "s" on those). To be consistent with matplotlib.pyplot.pie() you must use labels and colors.

If you want to hide wedge labels, specify labels=None. If fontsize is specified, the value will be applied to wedge labels. Also, other keywords supported by matplotlib.pyplot.pie() can be used.

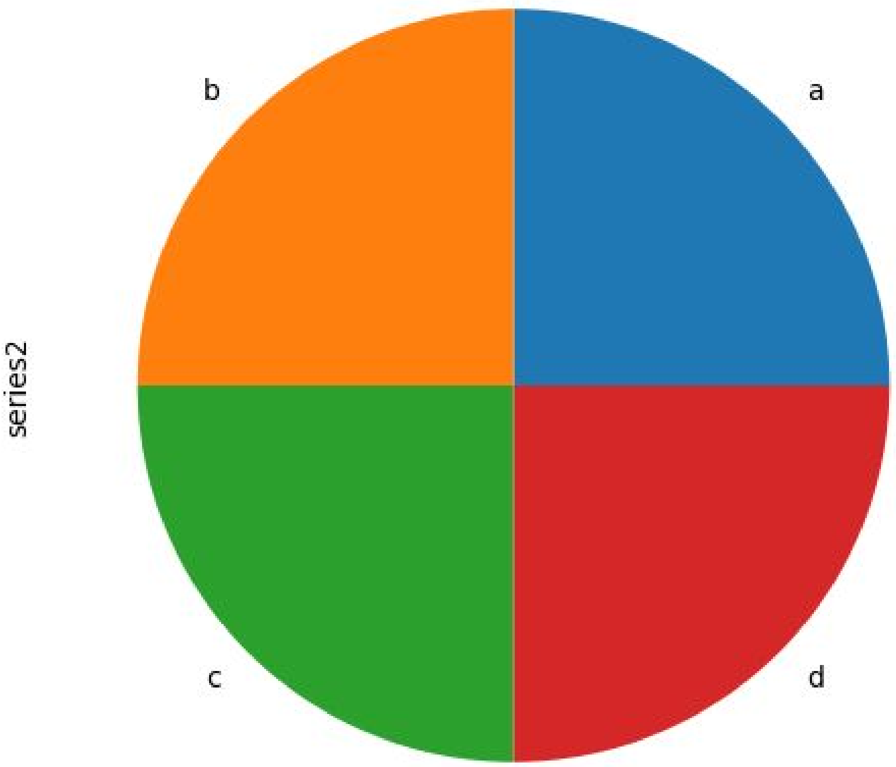
```
In [88]: series.plot.pie(
   ....:     labels=["AA", "BB", "CC", "DD"],
   ....:     colors=["r", "g", "b", "c"],
   ....:     autopct="%.2f",
   ....:     fontsize=20,
   ....:     figsize=(6, 6),
   ....: );
   ....:
```



If you pass values whose sum total is less than 1.0, matplotlib draws a semicircle.

```
In [89]: series = pd.Series([0.1] * 4, index=["a", "b", "c", "d"], name="series2")
```

```
In [90]: series.plot.pie(figsize=(6, 6));
```



See the matplotlib pie documentation for more.

Plotting with missing data

pandas tries to be pragmatic about plotting DataFrames or Series that contain missing data. Missing values are dropped, left out, or filled depending on the plot type.

Plot Type	NaN Handling
Line	Leave gaps at NaNs
Line (stacked)	Fill 0’s
Bar	Fill 0’s
Scatter	Drop NaNs
Histogram	Drop NaNs (column-wise)
Box	Drop NaNs (column-wise)
Area	Fill 0’s
KDE	Drop NaNs (column-wise)
Hexbin	Drop NaNs
Pie	Fill 0’s

If any of these defaults are not what you want, or if you want to be explicit about how missing values are handled, consider using fillna() or dropna() before plotting.

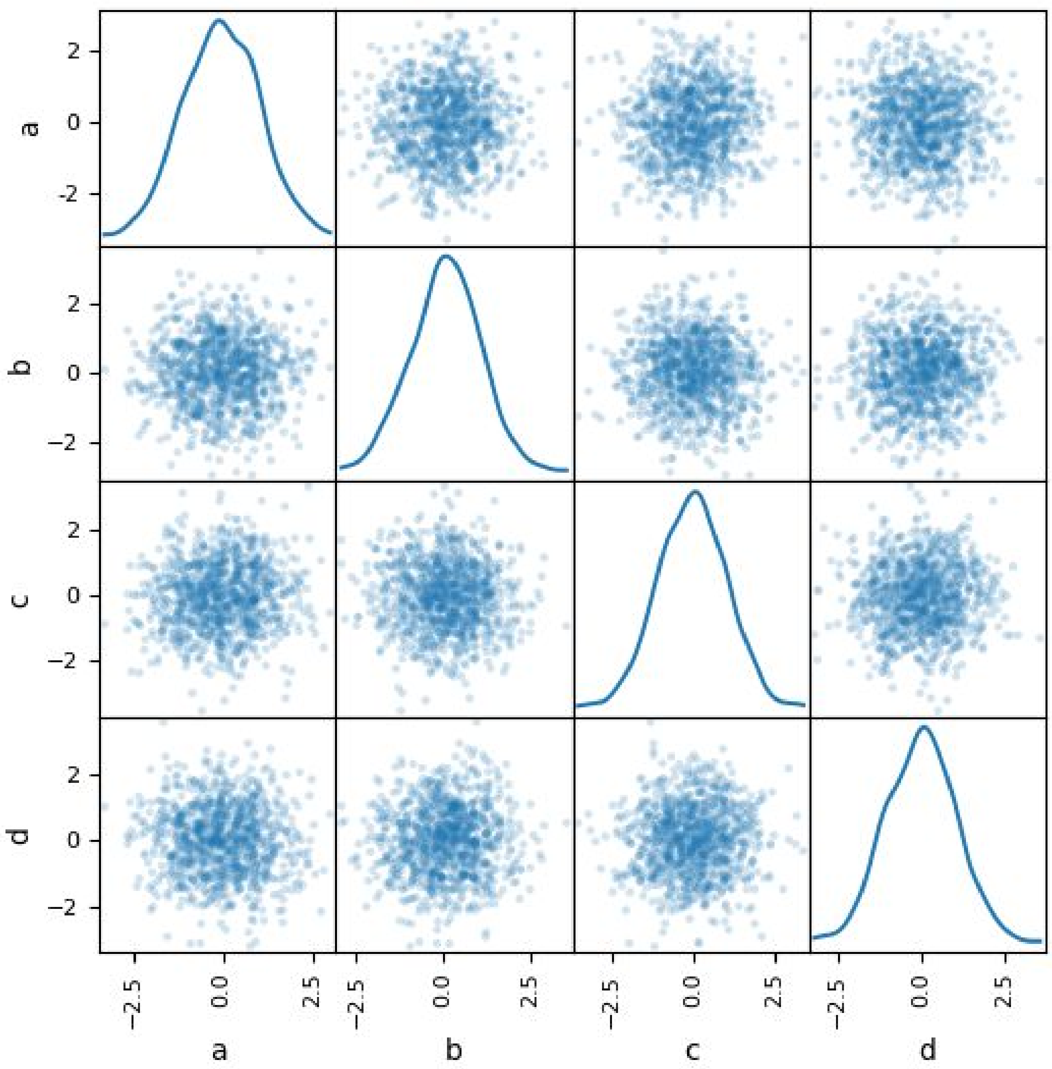
Plotting tools

These functions can be imported from pandas.plotting and take a Series or DataFrame as an argument.

Scatter matrix plot

You can create a scatter plot matrix using the scatter_matrix method in pandas.plotting:

```
In [91]: from pandas.plotting import scatter_matrix

In [92]: df = pd.DataFrame(np.random.randn(1000, 4), columns=["a", "b", "c", "d"])

In [93]: scatter_matrix(df, alpha=0.2, figsize=(6, 6), diagonal="kde");
```

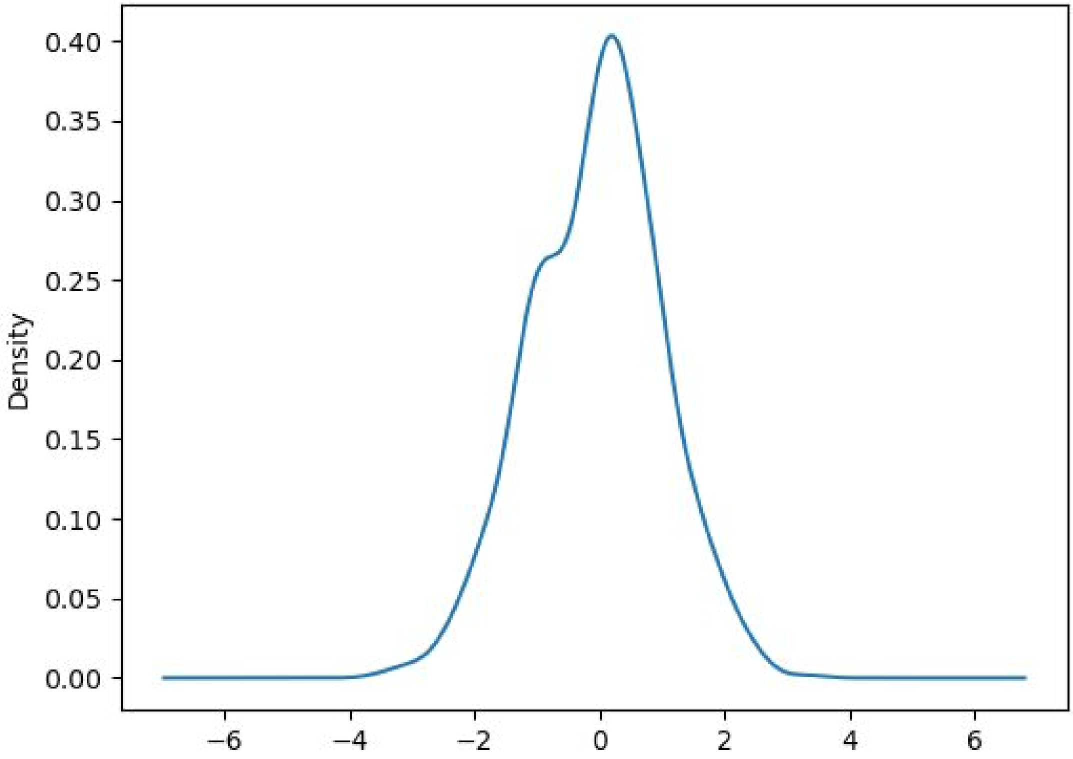


Density plot

You can create density plots using the Series.plot.kde() and DataFrame.plot.kde() methods.

```
In [94]: ser = pd.Series(np.random.randn(1000))
```

```
In [95]: ser.plot.kde();
```



Andrews curves

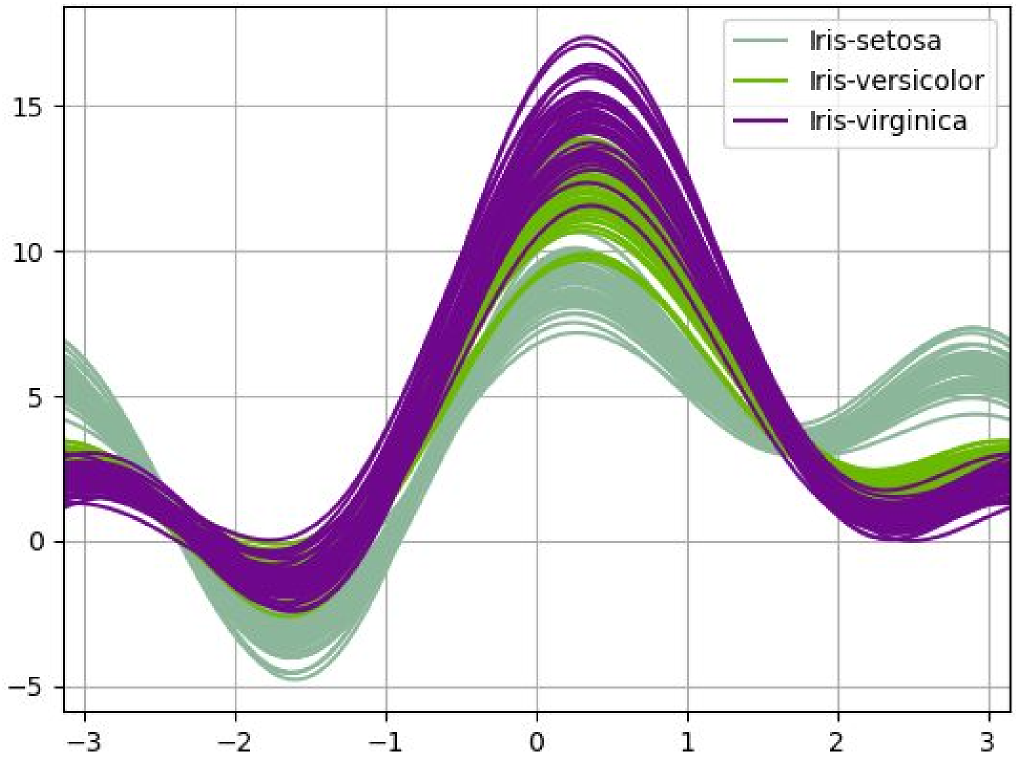
Andrews curves allow one to plot multivariate data as a large number of curves that are created using the attributes of samples as coefficients for Fourier series, see the Wikipedia entry for more information. By coloring these curves differently for each class it is possible to visualize data clustering. Curves belonging to samples of the same class will usually be closer together and form larger structures.

Note: The “Iris” dataset is available here.

```
In [96]: from pandas.plotting import andrews_curves
```

```
In [97]: data = pd.read_csv("data/iris.data")
In [98]: plt.figure();
```

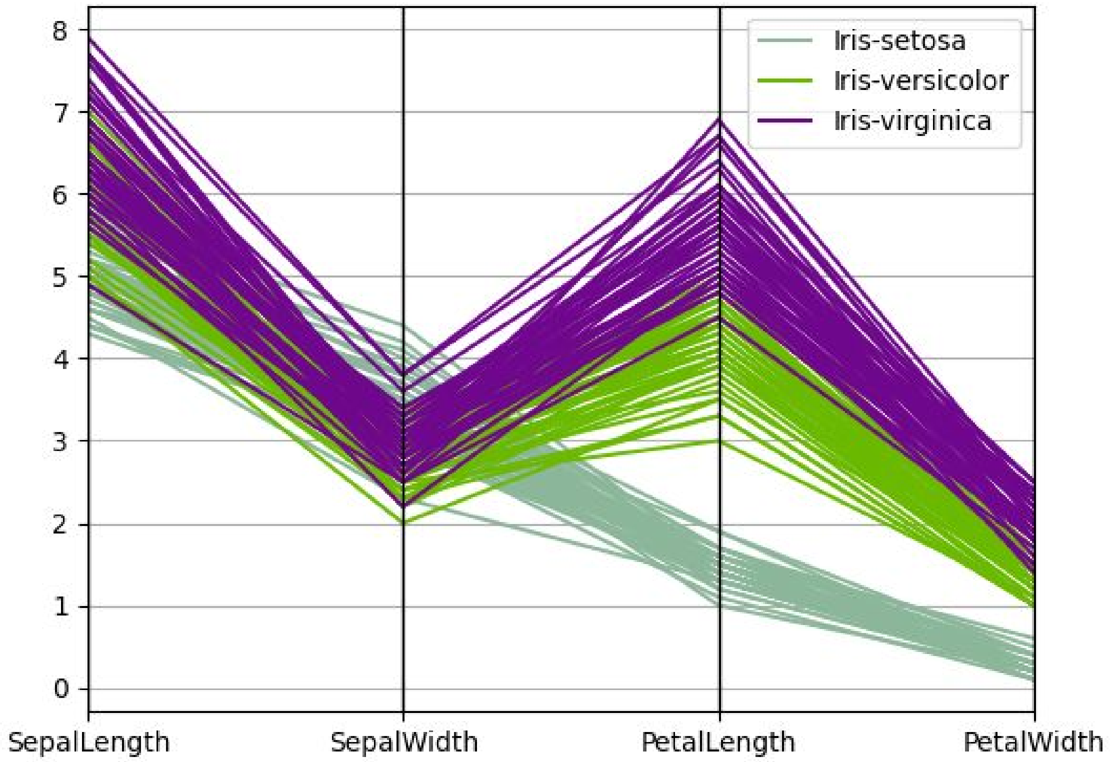
```
In [99]: andrews_curves(data, "Name");
```



Parallel coordinates

Parallel coordinates is a plotting technique for plotting multivariate data, see the Wikipedia entry for an introduction. Parallel coordinates allows one to see clusters in data and to estimate other statistics visually. Using parallel coordinates points are represented as connected line segments. Each vertical line represents one attribute. One set of connected line segments represents one data point. Points that tend to cluster will appear closer together.

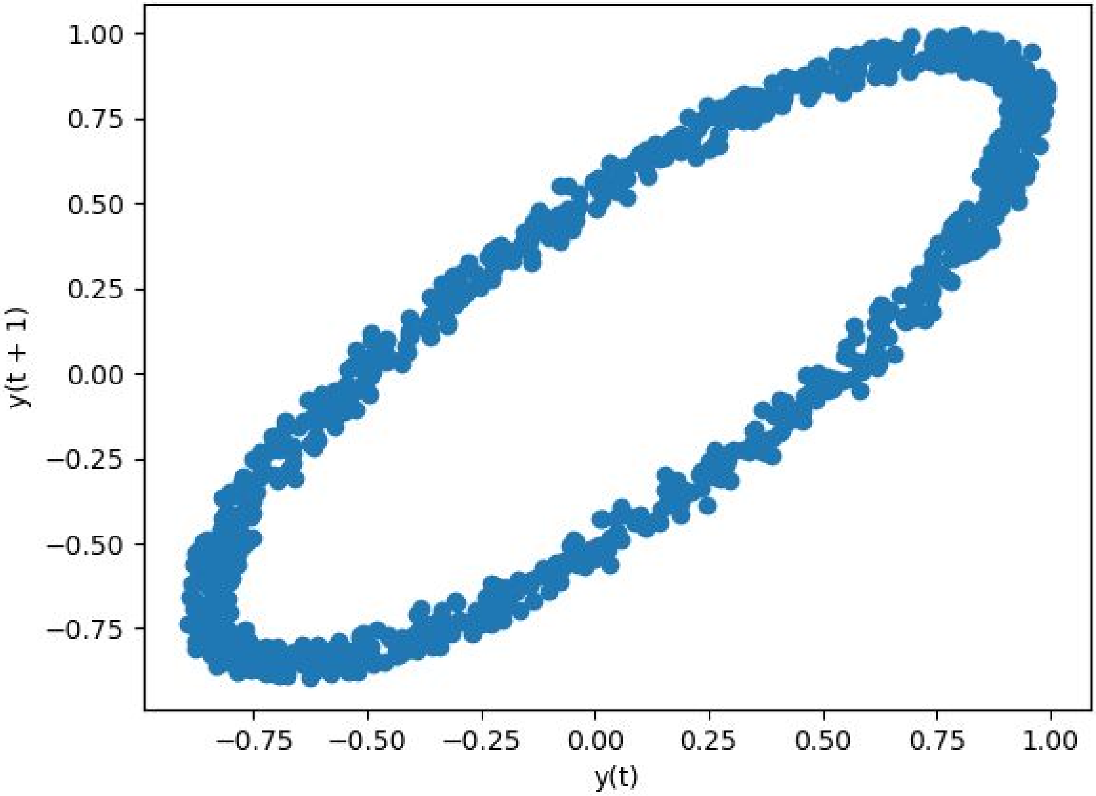
```
In [100]: from pandas.plotting import parallel_coordinates

In [101]: data = pd.read_csv("data/iris.data")

In [102]: plt.figure();

In [103]: parallel_coordinates(data, "Name");
```



Lag plot

Lag plots are used to check if a data set or time series is random. Random data should not exhibit any structure in the lag plot. Non-random structure implies that the underlying data are not random. The lag argument may be passed, and when lag=1 the plot is essentially data[:-1] vs. data[1:].

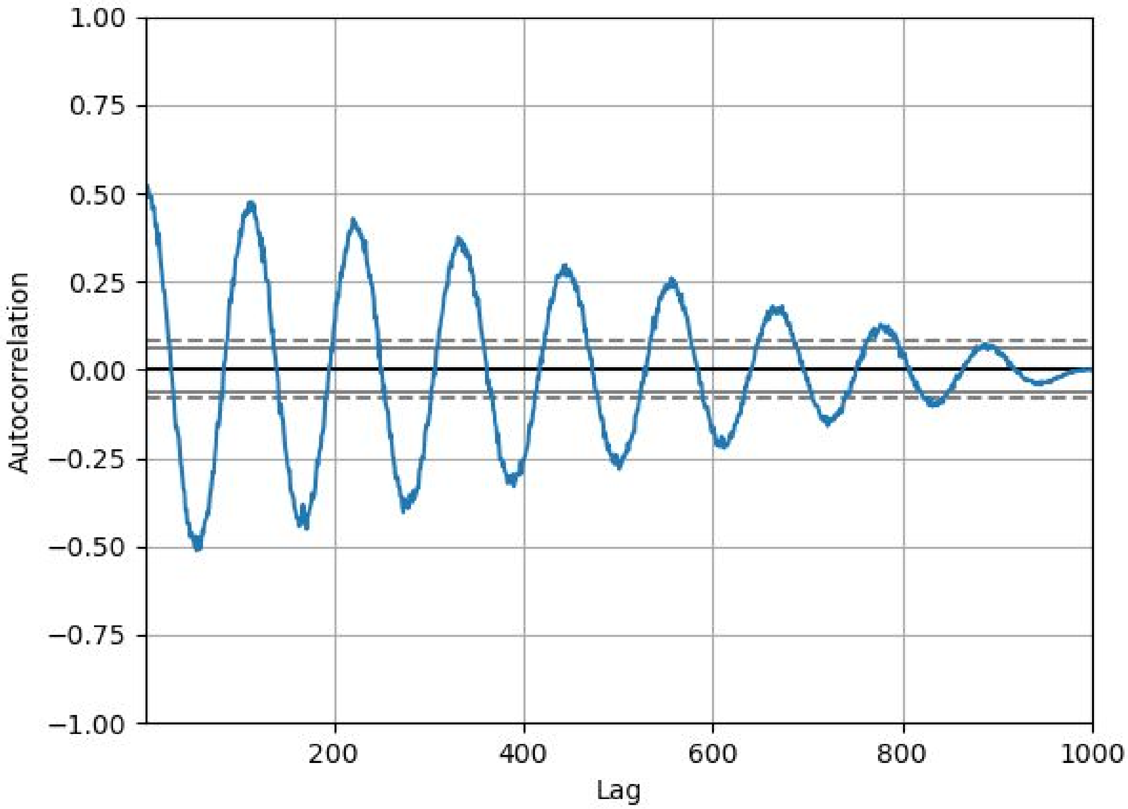
```
In [104]: from pandas.plotting import lag_plot

In [105]: plt.figure();


In [106]: spacing = np.linspace(-99 * np.pi, 99 * np.pi, num=1000)

In [107]: data = pd.Series(0.1 * np.random.rand(1000) + 0.9 * np.sin(spacing))

In [108]: lag_plot(data);
```



Autocorrelation plot

Autocorrelation plots are often used for checking randomness in time series. This is done by computing autocorrelations for data values at varying time lags. If time series is random, such autocorrelations should be near zero for any and all time-lag separations. If time series is non-random then one or more of the autocorrelations will be significantly non-zero. The horizontal lines displayed in the plot correspond to 95% and 99% confidence bands. The dashed line is 99% confidence band. See the Wikipedia entry for more about autocorrelation plots.

```
In [109]: from pandas.plotting import autocorrelation_plot

In [110]: plt.figure();

In [111]: spacing = np.linspace(-9 * np.pi, 9 * np.pi, num=1000)

In [112]: data = pd.Series(0.7 * np.random.rand(1000) + 0.3 * np.sin(spacing))

In [113]: autocorrelation_plot(data);
```



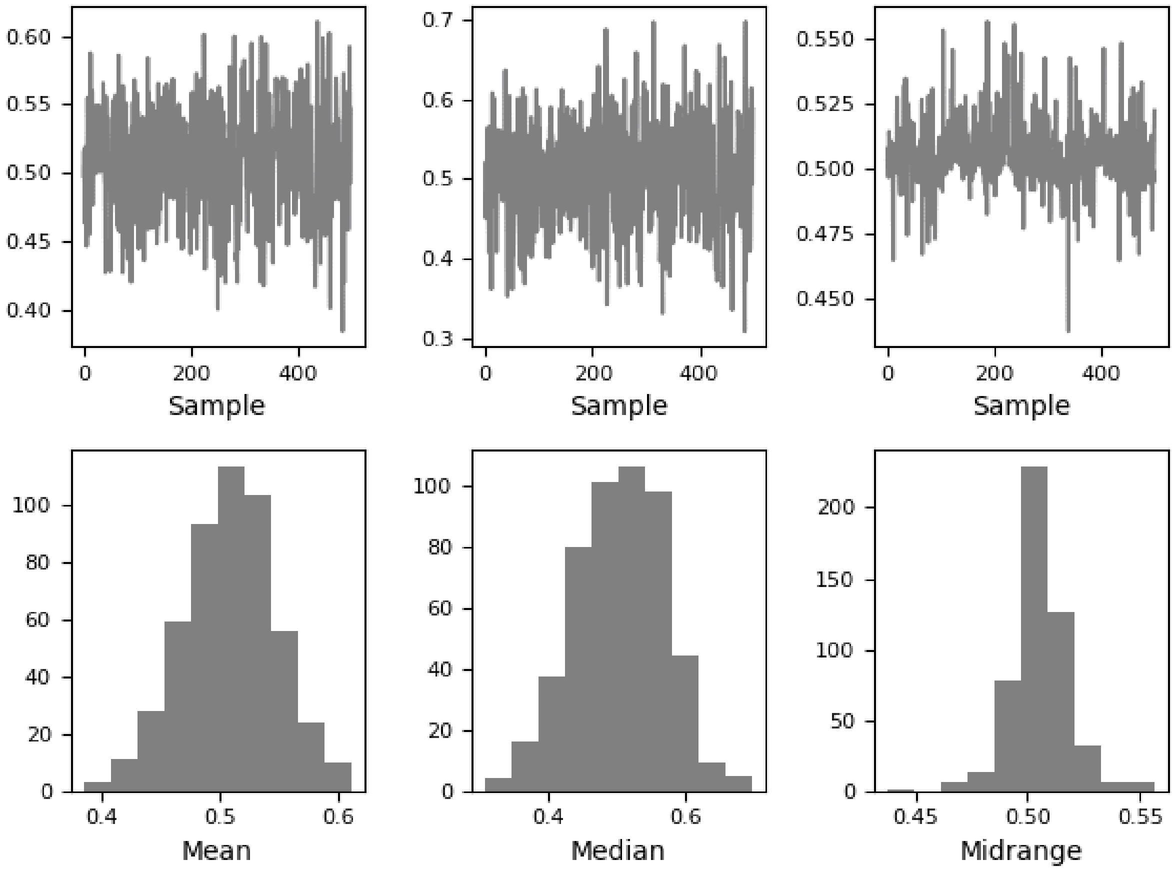
Bootstrap plot

Bootstrap plots are used to visually assess the uncertainty of a statistic, such as mean, median, midrange, etc. A random subset of a specified size is selected from a data set, the statistic in question is computed for this subset and the process is repeated a specified number of times. Resulting plots and histograms are what constitutes the bootstrap plot.

```
In [114]: from pandas.plotting import bootstrap_plot
```

```
In [115]: data = pd.Series(np.random.rand(1000))
```

```
In [116]: bootstrap_plot(data, size=50, samples=500, color="grey");
```

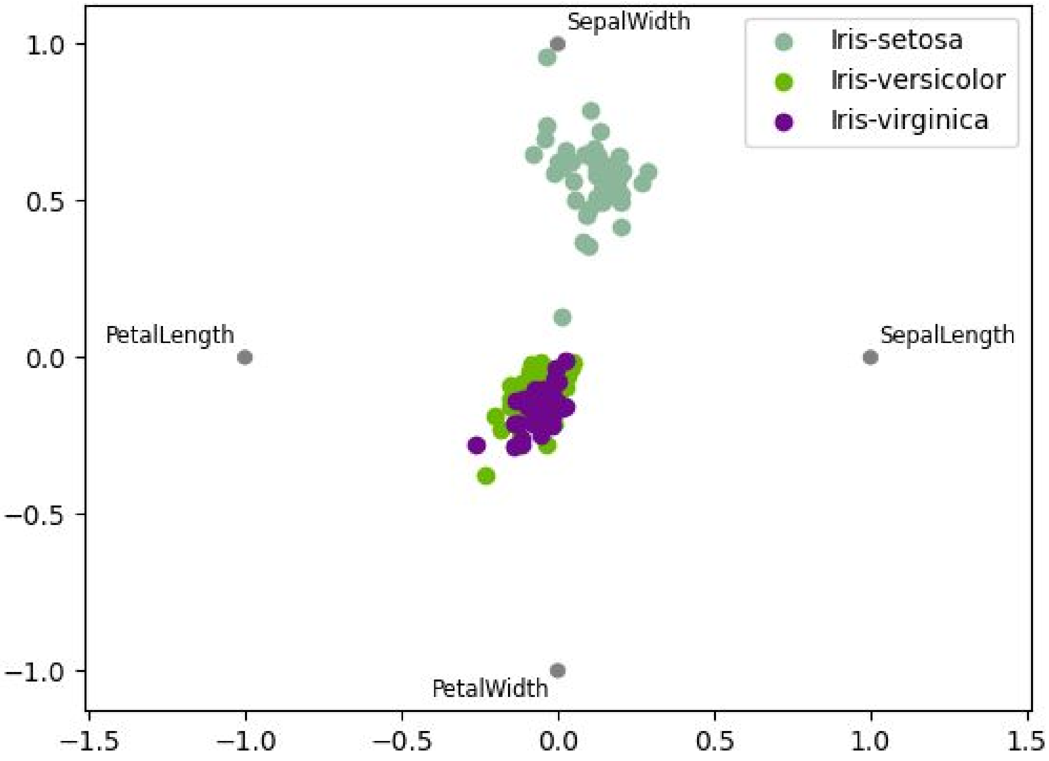


RadViz

RadViz is a way of visualizing multi-variate data. It is based on a simple spring tension minimization algorithm. Basically you set up a bunch of points in a plane. In our case they are equally spaced on a unit circle. Each point represents a single attribute. You then pretend that each sample in the data set is attached to each of these points by a spring, the stiffness of which is proportional to the numerical value of that attribute (they are normalized to unit interval). The point in the plane, where our sample settles to (where the forces acting on our sample are at an equilibrium) is where a dot representing our sample will be drawn. Depending on which class that sample belongs it will be colored differently. See the R package Radviz for more information.

Note: The “Iris” dataset is available here.

```
In [117]: from pandas.plotting import radviz

In [118]: data = pd.read_csv("data/iris.data")

In [119]: plt.figure();

In [120]: radviz(data, "Name");
```



Plot formatting

Setting the plot style

From version 1.5 and up, matplotlib offers a range of pre-configured plotting styles. Setting the style can be used to easily give plots the general look that you want. Setting the style is as easy as calling matplotlib.style.use(my_-

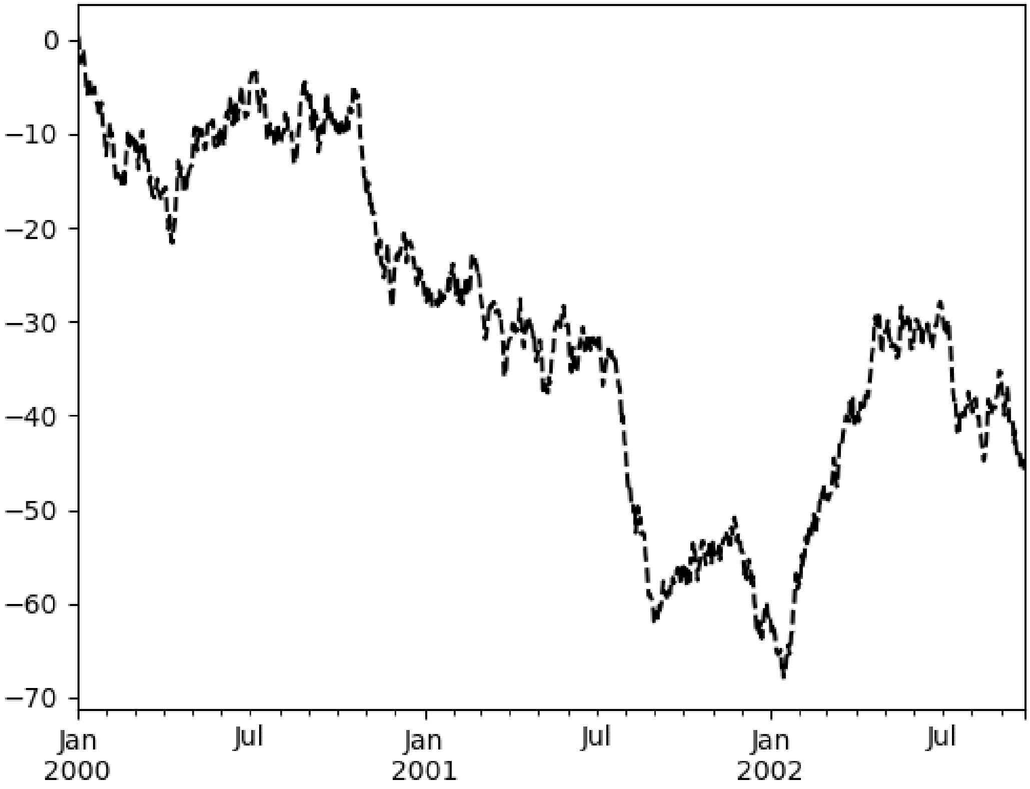
plot_style) before creating your plot. For example you could write matplotlib.style.use('ggplot') for ggplot-style plots.

You can see the various available style names at matplotlib.style.available and it's very easy to try them out.

General plot style arguments

Most plotting methods have a set of keyword arguments that control the layout and formatting of the returned plot:

```
In [121]: plt.figure();

In [122]: ts.plot(style="k--", label="Series");
```

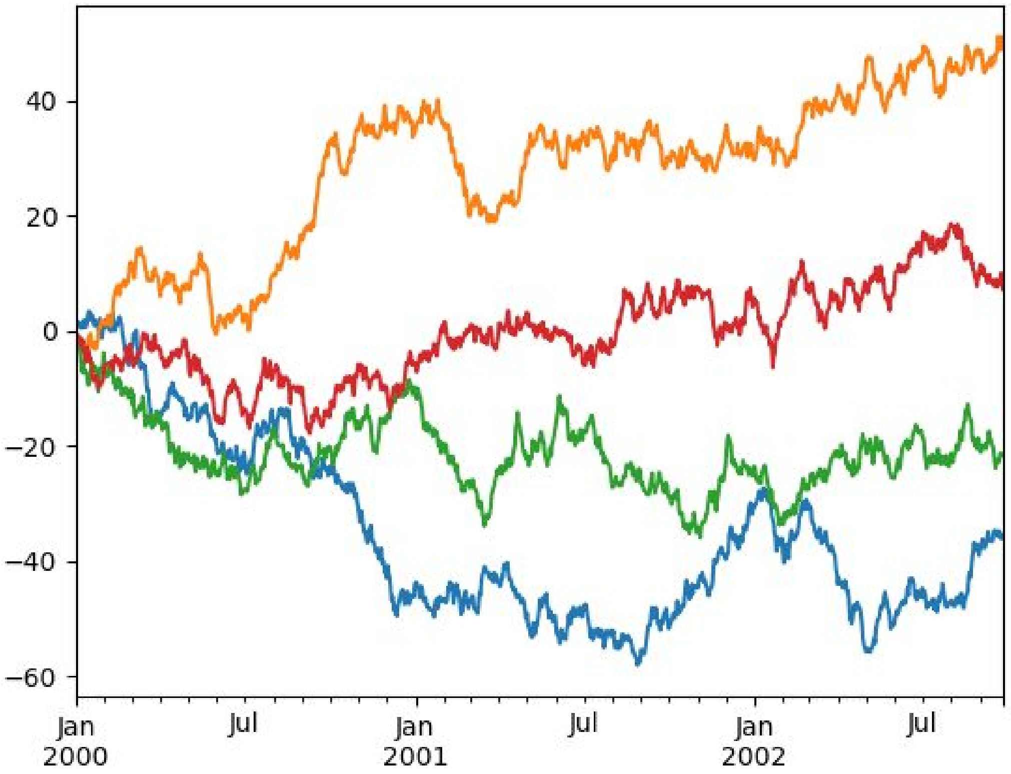


For each kind of plot (e.g. line, bar, scatter) any additional arguments keywords are passed along to the corresponding matplotlib function (ax.plot(), ax.bar(), ax.scatter()). These can be used to control additional styling, beyond what pandas provides.

Controlling the legend

You may set the legend argument to False to hide the legend, which is shown by default.

```
In [123]: df = pd.DataFrame(np.random.randn(1000, 4), index=ts.index, columns=list("ABCD"))

In [124]: df = df.cumsum()

In [125]: df.plot(legend=False);
```

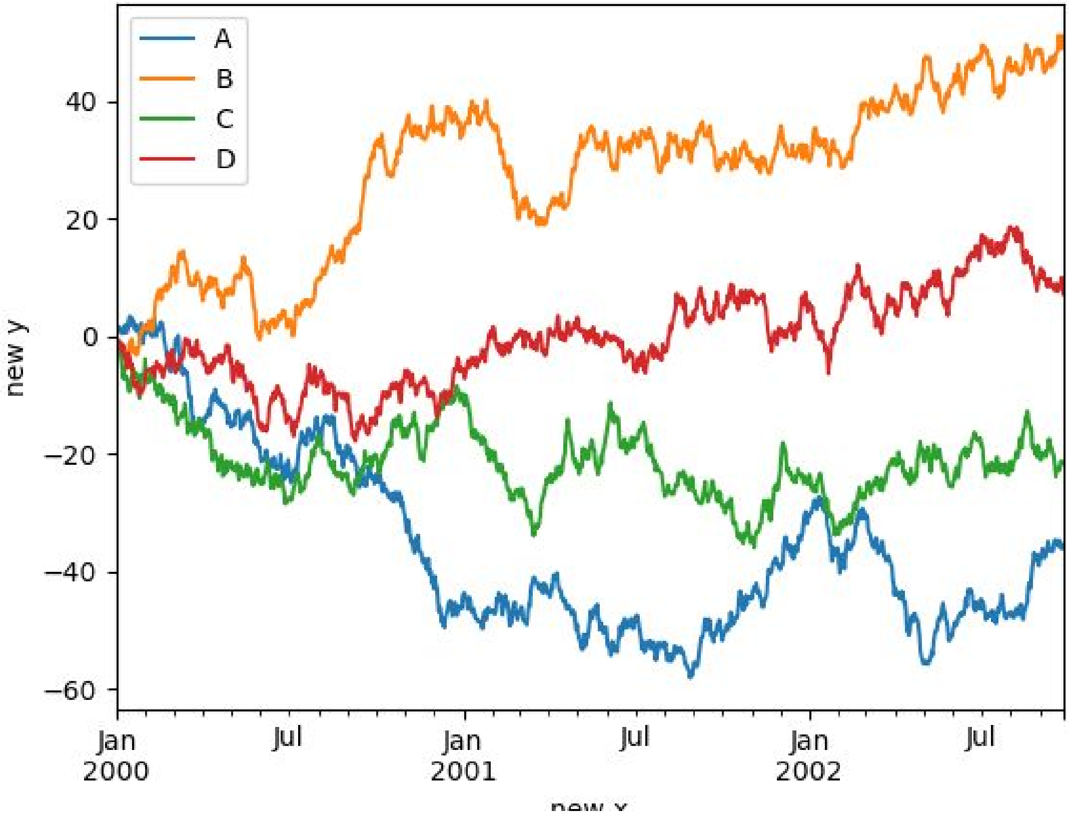


Controlling the labels

New in version 1.1.0.

You may set the xlabel and ylabel arguments to give the plot custom labels for x and y axis. By default, pandas will pick up index name as xlabel, while leaving it empty for ylabel.

```
In [126]: df.plot();

In [127]: df.plot(xlabel="new x", ylabel="new y");
```

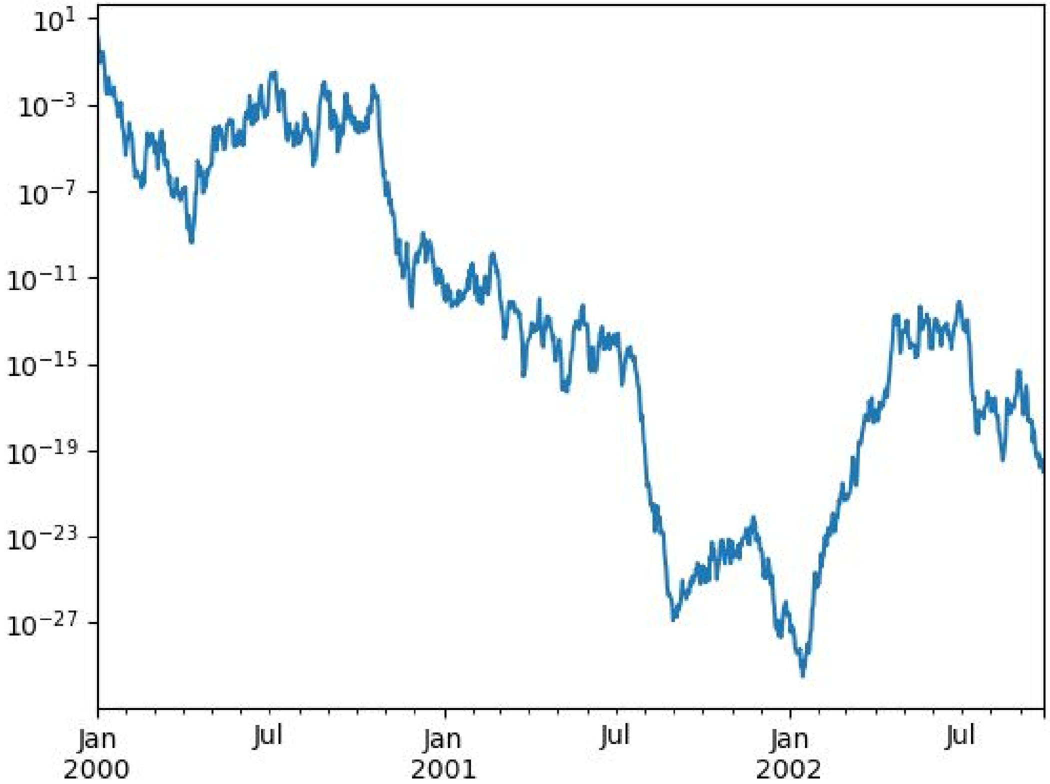


Scales

You may pass logy to get a log-scale Y axis.

```
In [128]: ts = pd.Series(np.random.randn(1000), index=pd.date_range("1/1/2000", periods=1000))

In [129]: ts = np.exp(ts.cumsum())
```

```
In [130]: ts.plot(logy=True);
```

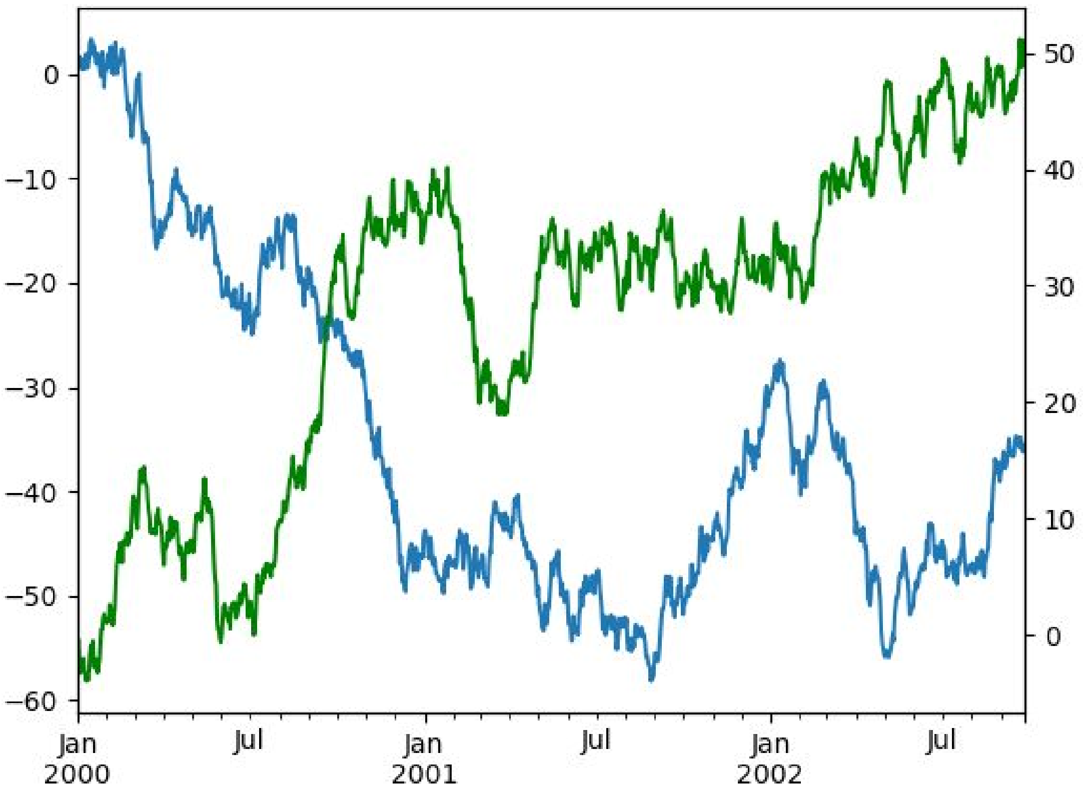


See also the logx and loglog keyword arguments.

Plotting on a secondary y-axis

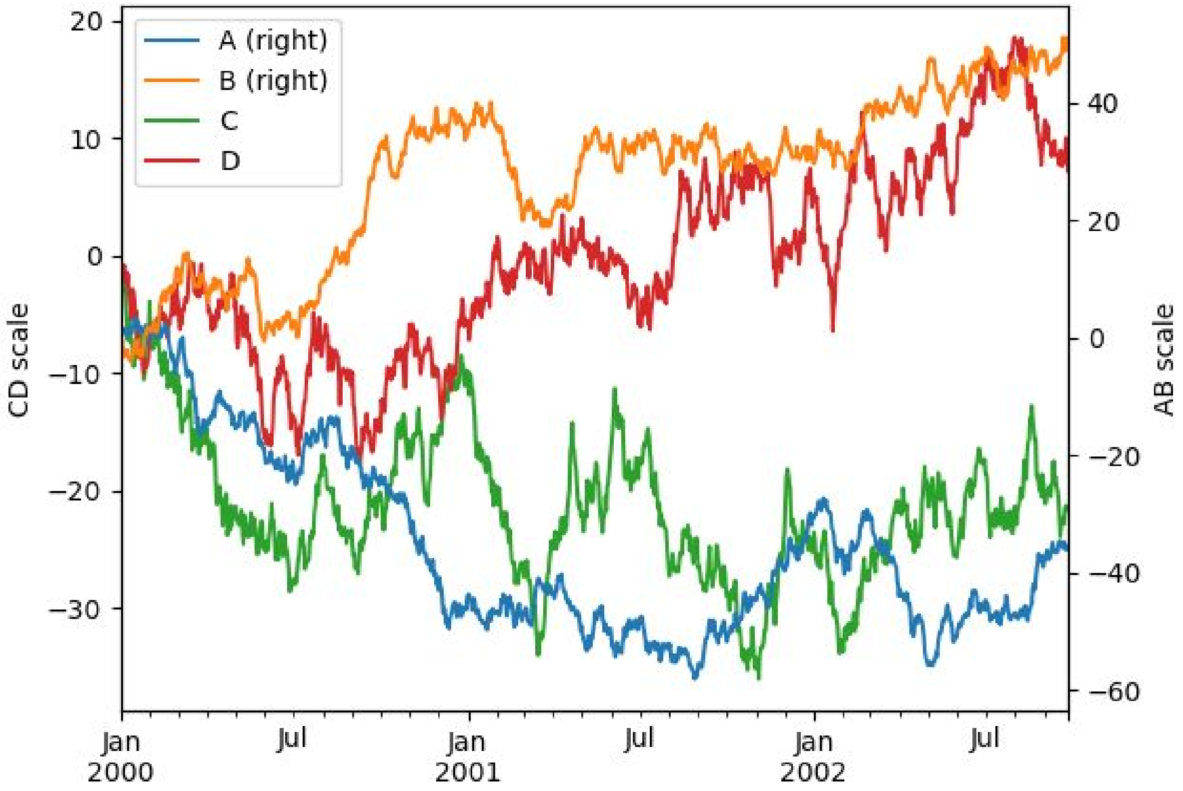
To plot data on a secondary y-axis, use the secondary_y keyword:

```
In [131]: df["A"].plot();

In [132]: df["B"].plot(secondary_y=True, style="g");
```



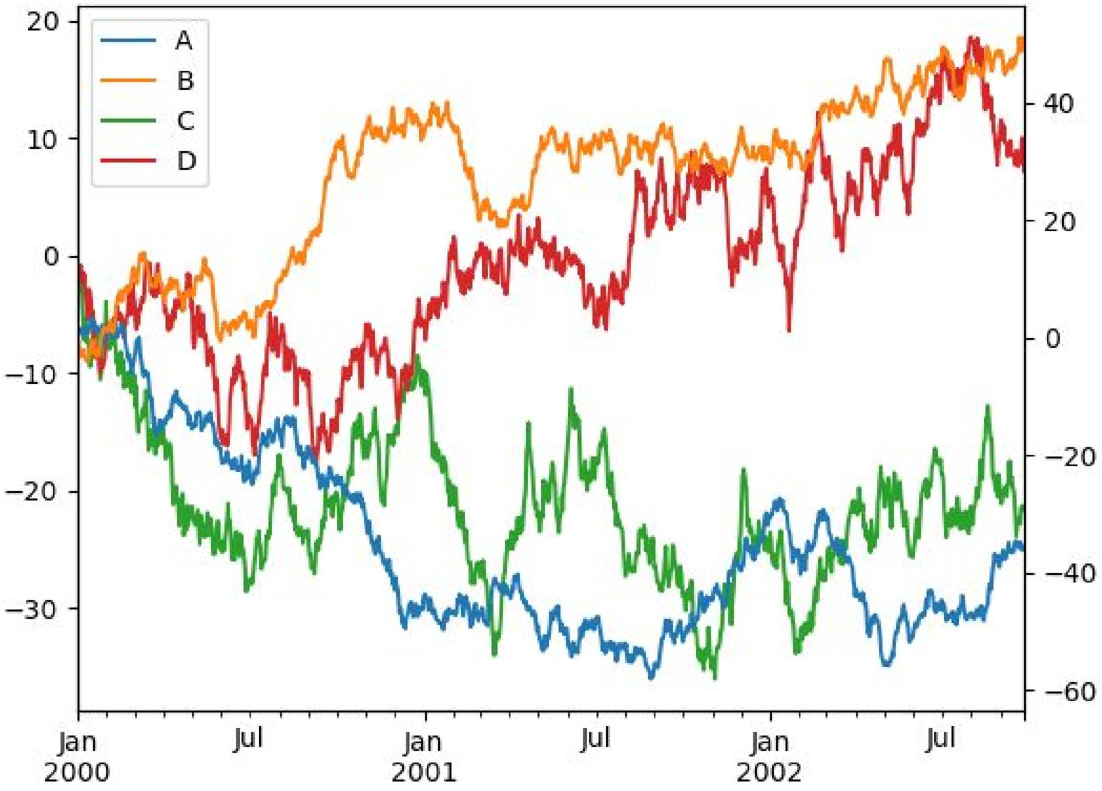
To plot some columns in a DataFrame, give the column names to the secondary_y keyword:

```
In [133]: plt.figure();

In [134]: ax = df.plot(secondary_y=["A", "B"])

In [135]: ax.set_ylabel("CD scale");

In [136]: ax.right_ax.set_ylabel("AB scale");
```



Note that the columns plotted on the secondary y-axis is automatically marked with “(right)” in the legend. To turn off the automatic marking, use the mark_right=False keyword:

```
In [137]: plt.figure();

In [138]: df.plot(secondary_y=["A", "B"], mark_right=False);
```



Custom formatters for timeseries plots

Changed in version 1.0.0.

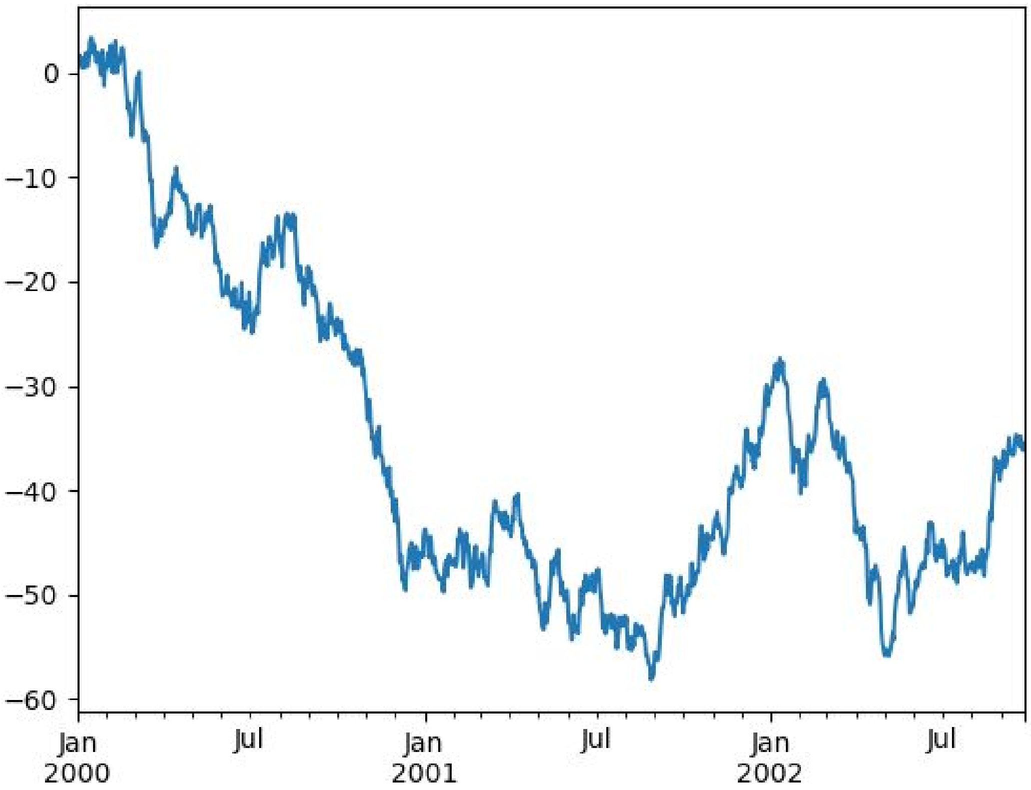
pandas provides custom formatters for timeseries plots. These change the formatting of the axis labels for dates and times. By default, the custom formatters are applied only to plots created by pandas with DataFrame.plot() or Series.plot(). To have them apply to all plots, including those made by matplotlib, set the option pd.options.plotting.matplotlib.register_converters = True or use pandas.plotting.register_matplotlib_converters().

Suppressing tick resolution adjustment

pandas includes automatic tick resolution adjustment for regular frequency time-series data. For limited cases where pandas cannot infer the frequency information (e.g., in an externally created twinx), you can choose to suppress this behavior for alignment purposes.

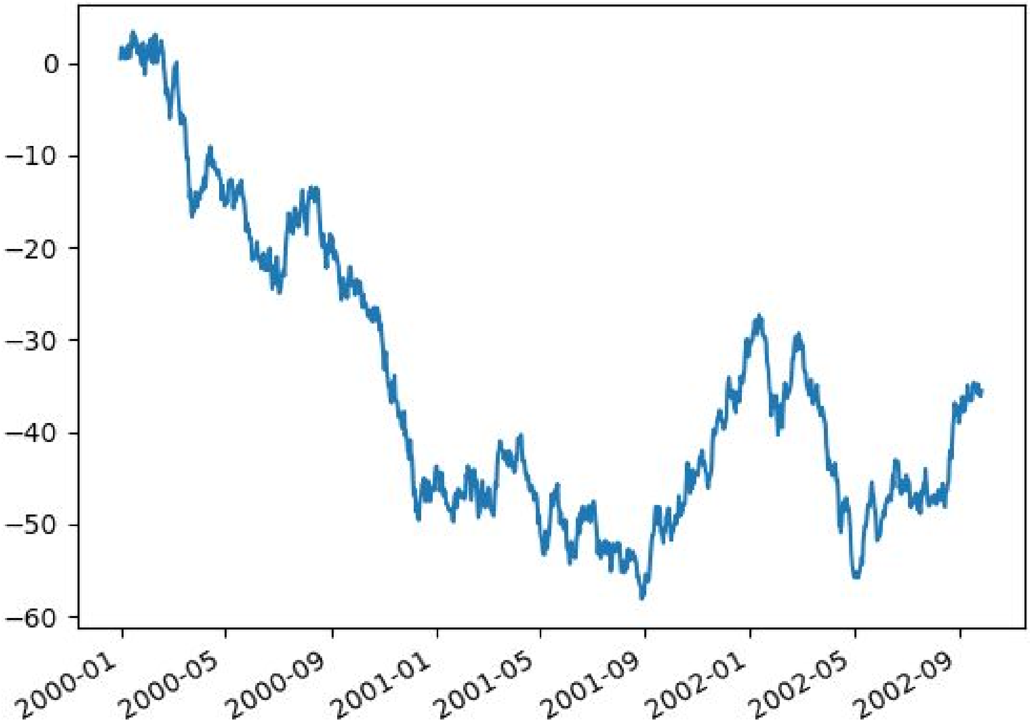
Here is the default behavior, notice how the x-axis tick labeling is performed:

```
In [139]: plt.figure();

In [140]: df["A"].plot();
```



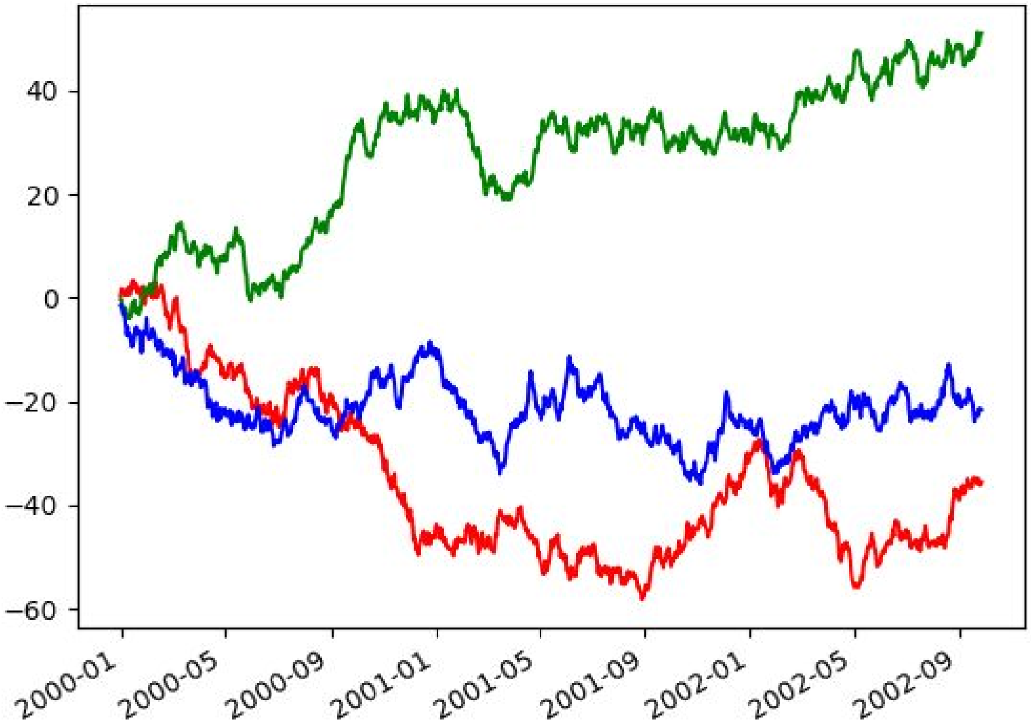
Using the x_compat parameter, you can suppress this behavior:

```
In [141]: plt.figure();

In [142]: df["A"].plot(x_compat=True);
```



If you have more than one plot that needs to be suppressed, the use method in pandas.plotting.plot_params can be used in a with statement:

```
In [143]: plt.figure();

In [144]: with pd.plotting.plot_params.use("x_compat", True):

   .....:     df["A"].plot(color="r")

   .....:     df["B"].plot(color="g")

   .....:     df["C"].plot(color="b")

   .....:
```



Automatic date tick adjustment

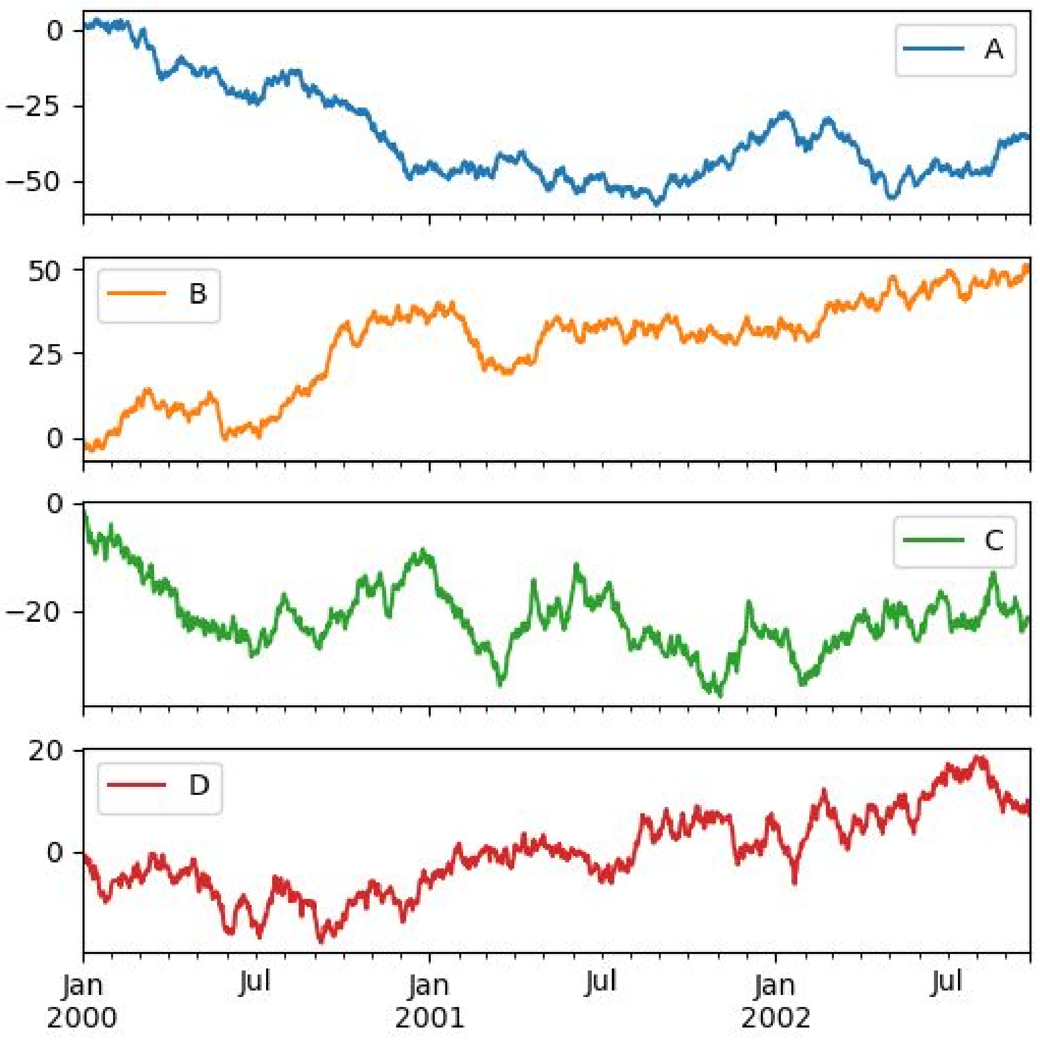
TimedeltaIndex now uses the native matplotlib tick locator methods, it is useful to call the automatic date tick adjustment from matplotlib for figures whose ticklabels overlap.

See the autofmt_xdate method and the matplotlib documentation for more.

Subplots

Each Series in a DataFrame can be plotted on a different axis with the subplots keyword:

```
In [145]: df.plot(subplots=True, figsize=(6, 6));
```

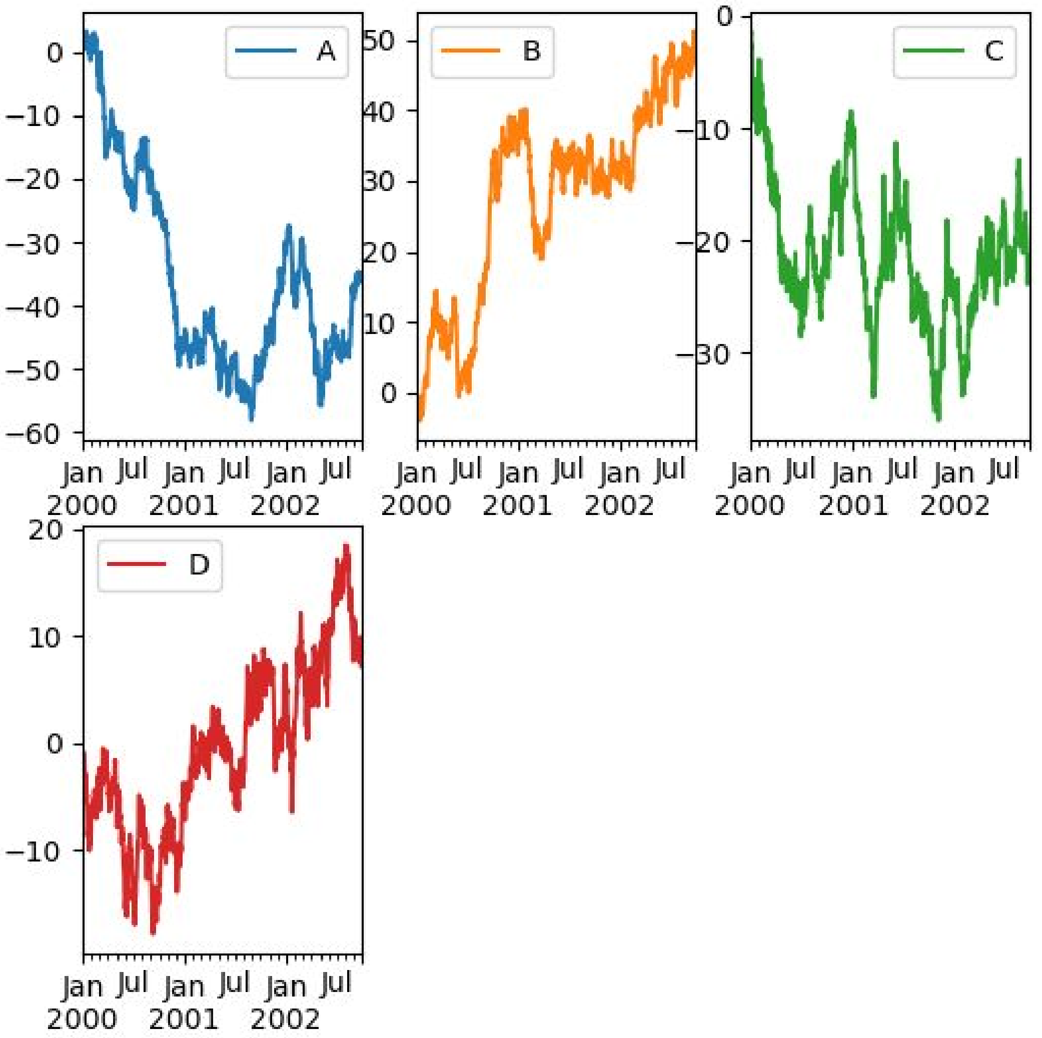


Using layout and targeting multiple axes

The layout of subplots can be specified by the layout keyword. It can accept (rows, columns). The layout keyword can be used in hist and boxplot also. If the input is invalid, a ValueError will be raised.

The number of axes which can be contained by rows x columns specified by layout must be larger than the number of required subplots. If layout can contain more axes than required, blank axes are not drawn. Similar to a NumPy array’s reshape method, you can use -1 for one dimension to automatically calculate the number of rows or columns needed, given the other.

```
In [146]: df.plot(subplots=True, layout=(2, 3), figsize=(6, 6), sharex=False);
```



The above example is identical to using:

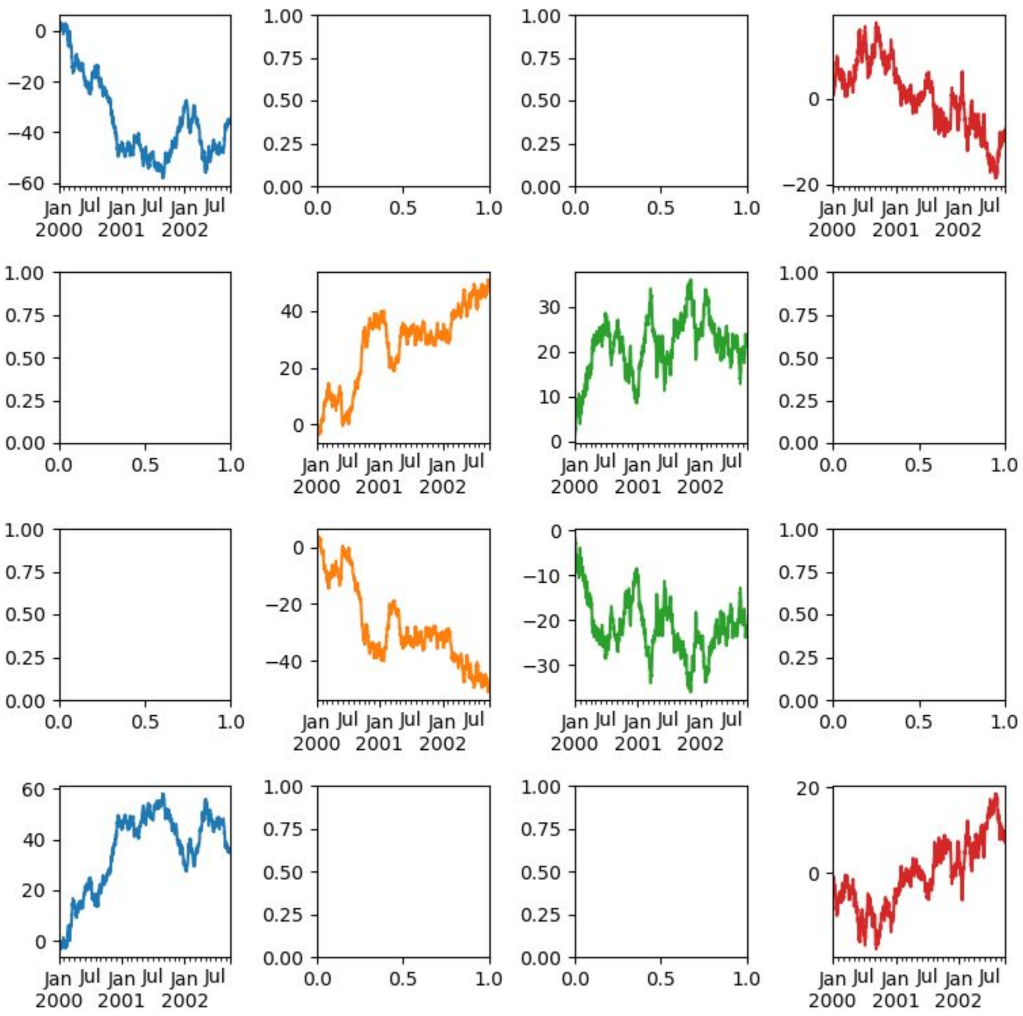
```
In [147]: df.plot(subplots=True, layout=(2, -1), figsize=(6, 6), sharex=False);
```

The required number of columns (3) is inferred from the number of series to plot and the given number of rows (2).

You can pass multiple axes created beforehand as list-like via ax keyword. This allows more complicated layouts. The passed axes must be the same number as the subplots being drawn.

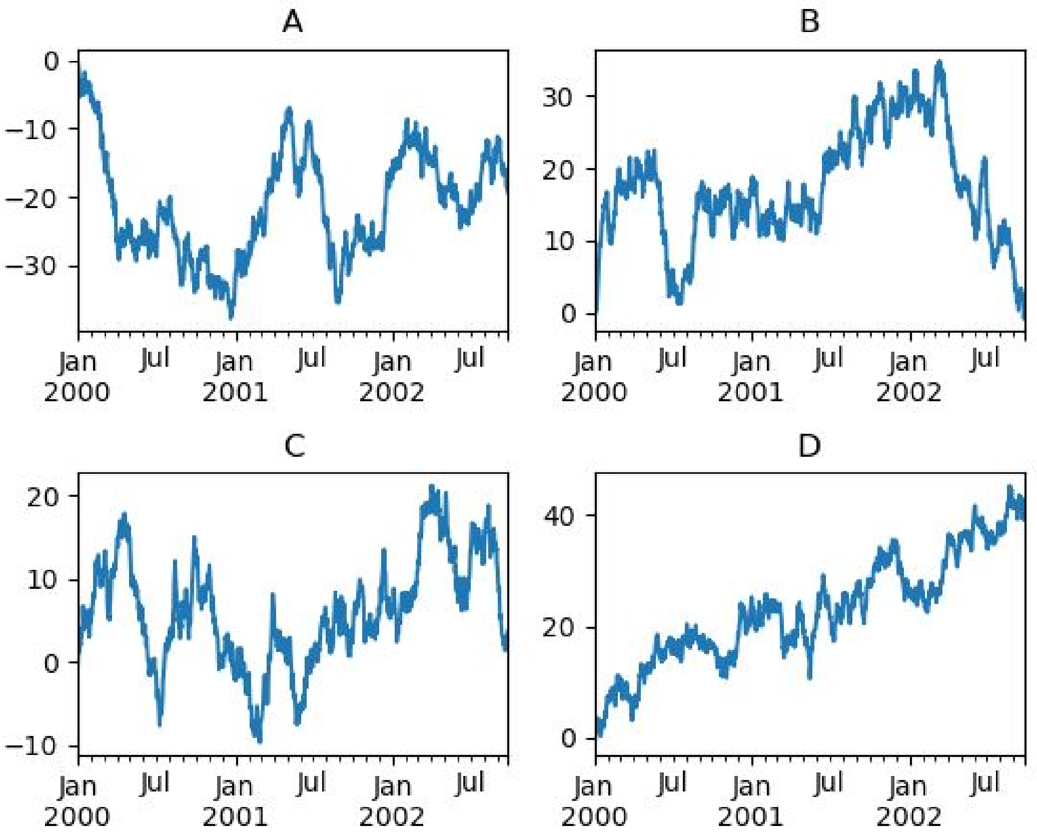
When multiple axes are passed via the ax keyword, layout, sharex and sharey keywords don't affect to the output. You should explicitly pass sharex=False and sharey=False, otherwise you will see a warning.

```
In [148]: fig, axes = plt.subplots(4, 4, figsize=(9, 9))

In [149]: plt.subplots_adjust(wspace=0.5, hspace=0.5)

In [150]: target1 = [axes[0][0], axes[1][1], axes[2][2], axes[3][3]]

In [151]: target2 = [axes[3][0], axes[2][1], axes[1][2], axes[0][3]]

In [152]: df.plot(subplots=True, ax=target1, legend=False, sharex=False, sharey=False);

In [153]: (-df).plot(subplots=True, ax=target2, legend=False, sharex=False, sharey=False);
```



Another option is passing an ax argument to Series.plot() to plot on a particular axis:

```
In [154]: fig, axes = plt.subplots(nrows=2, ncols=2)

In [155]: plt.subplots_adjust(wspace=0.2, hspace=0.5)

In [156]: df["A"].plot(ax=axes[0, 0]);
In [157]: axes[0, 0].set_title("A");

In [158]: df["B"].plot(ax=axes[0, 1]);

In [159]: axes[0, 1].set_title("B");

In [160]: df["C"].plot(ax=axes[1, 0]);

In [161]: axes[1, 0].set_title("C");

In [162]: df["D"].plot(ax=axes[1, 1]);

In [163]: axes[1, 1].set_title("D");
```



Plotting with error bars

Plotting with error bars is supported in DataFrame.plot() and Series.plot().

Horizontal and vertical error bars can be supplied to the xerr and yerr keyword arguments to plot(). The error values can be specified using a variety of formats:

- As a DataFrame or dict of errors with column names matching the columns attribute of the plotting DataFrame or matching the name attribute of the Series.
- As a str indicating which of the columns of plotting DataFrame contain the error values.
- As raw values (list, tuple, or np.ndarray). Must be the same length as the plotting DataFrame/Series.

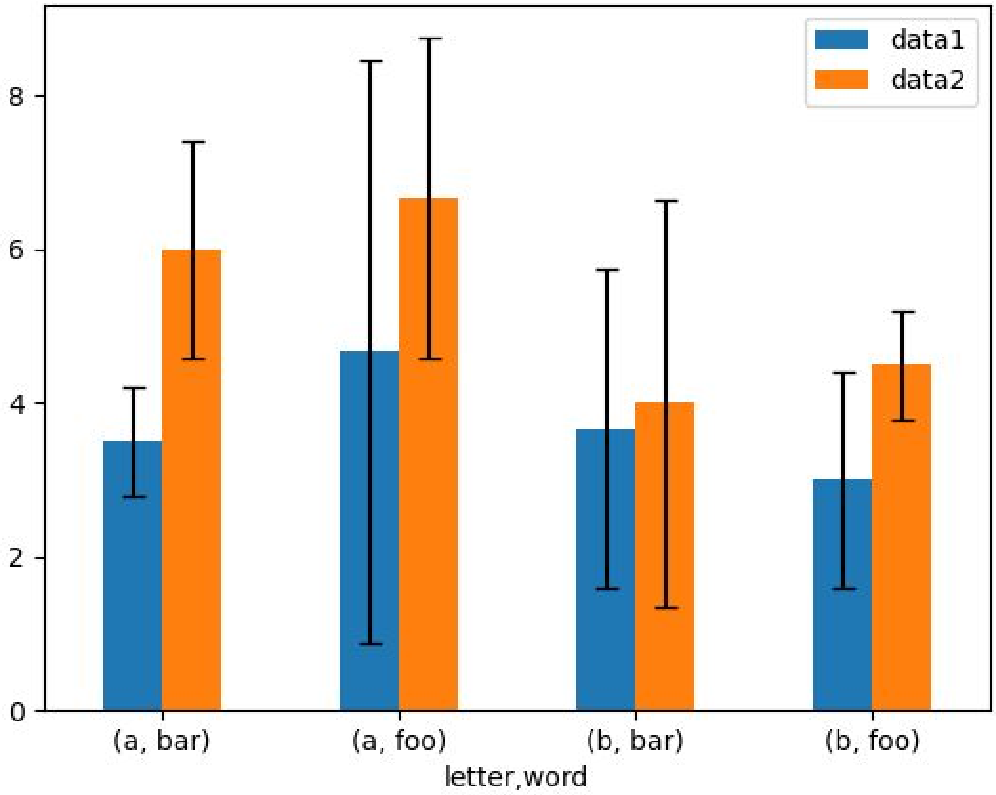
Here is an example of one way to easily plot group means with standard deviations from the raw data.

```
# Generate the data

In [164]: ix3 = pd.MultiIndex.from_arrays(
   .....:     [
   .....:         ["a", "a", "a", "a", "a", "b", "b", "b", "b", "b"],
   .....:         ["foo", "foo", "foo", "bar", "bar", "foo", "foo", "bar", "bar", "bar"],
   .....:     ],
   .....:     names=["letter", "word"],
   .....: )
   .....:


In [165]: df3 = pd.DataFrame(
   .....:     {
```

```
   .....:         "data1": [9, 3, 2, 4, 3, 2, 4, 6, 3, 2],
   .....:         "data2": [9, 6, 5, 7, 5, 4, 5, 6, 5, 1],
   .....:     },
   .....:     index=ix3,
   .....: )
   .....:

# Group by index labels and take the means and standard deviations
# for each group
In [166]: gp3 = df3.groupby(level=("letter", "word"))
In [167]: means = gp3.mean()
In [168]: errors = gp3.std()
In [169]: means
Out[169]:
                data1     data2
letter word
a      bar   3.500000  6.000000
       foo   4.666667  6.666667
b      bar   3.666667  4.000000
       foo   3.000000  4.500000

In [170]: errors
```

```
Out[170]:
             data1     data2
letter word
a      bar   0.707107  1.414214
       foo   3.785939  2.081666
b      bar   2.081666  2.645751
       foo   1.414214  0.707107
```

```
# Plot
In [171]: fig, ax = plt.subplots()


In [172]: means.plot.bar(yerr=errors, ax=ax, capsize=4, rot=0);
```



Asymmetrical error bars are also supported, however raw error values must be provided in this case. For a N length Series, a 2xN array should be provided indicating lower and upper (or left and right) errors. For a MxN DataFrame, asymmetrical errors should be in a Mx2xN array.

Here is an example of one way to plot the min/max range using asymmetrical error bars.

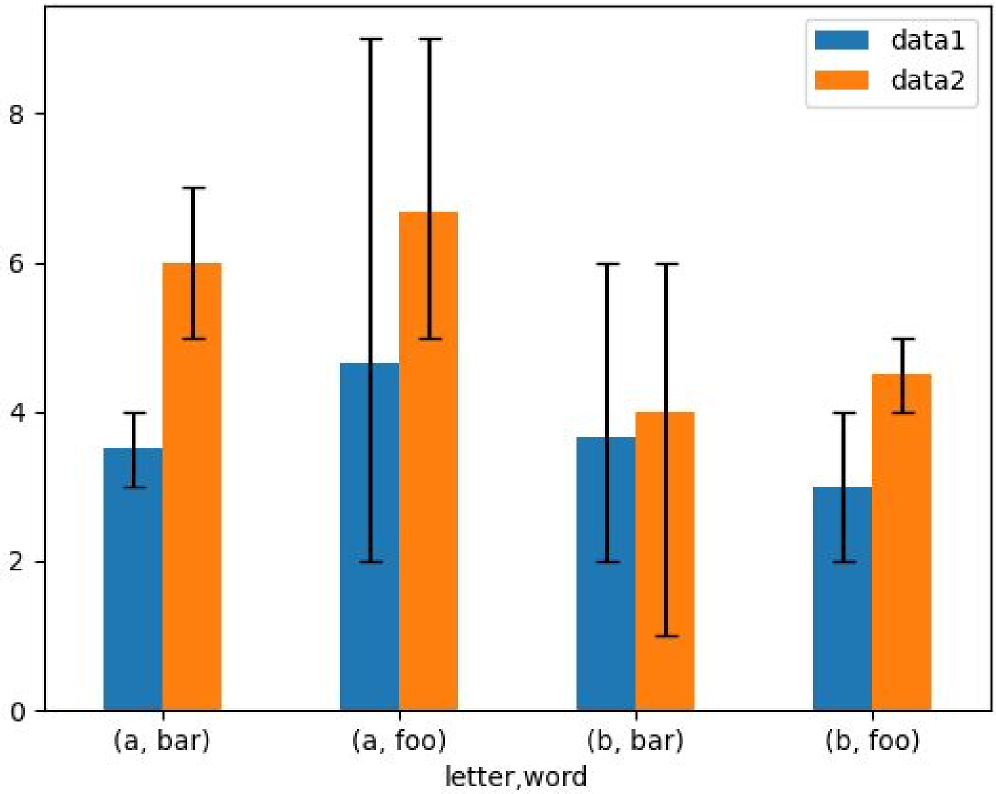
```
In [173]: mins = gp3.min()
```

```
In [174]: maxs = gp3.max()
```

```
# errors should be positive, and defined in the order of lower, upper
```

```
In [175]: errors = [[means[c] - mins[c], maxs[c] - means[c]] for c in df3.columns]
```

```
# Plot

In [176]: fig, ax = plt.subplots()

In [177]: means.plot.bar(yerr=errors, ax=ax, capsize=4, rot=0);
```



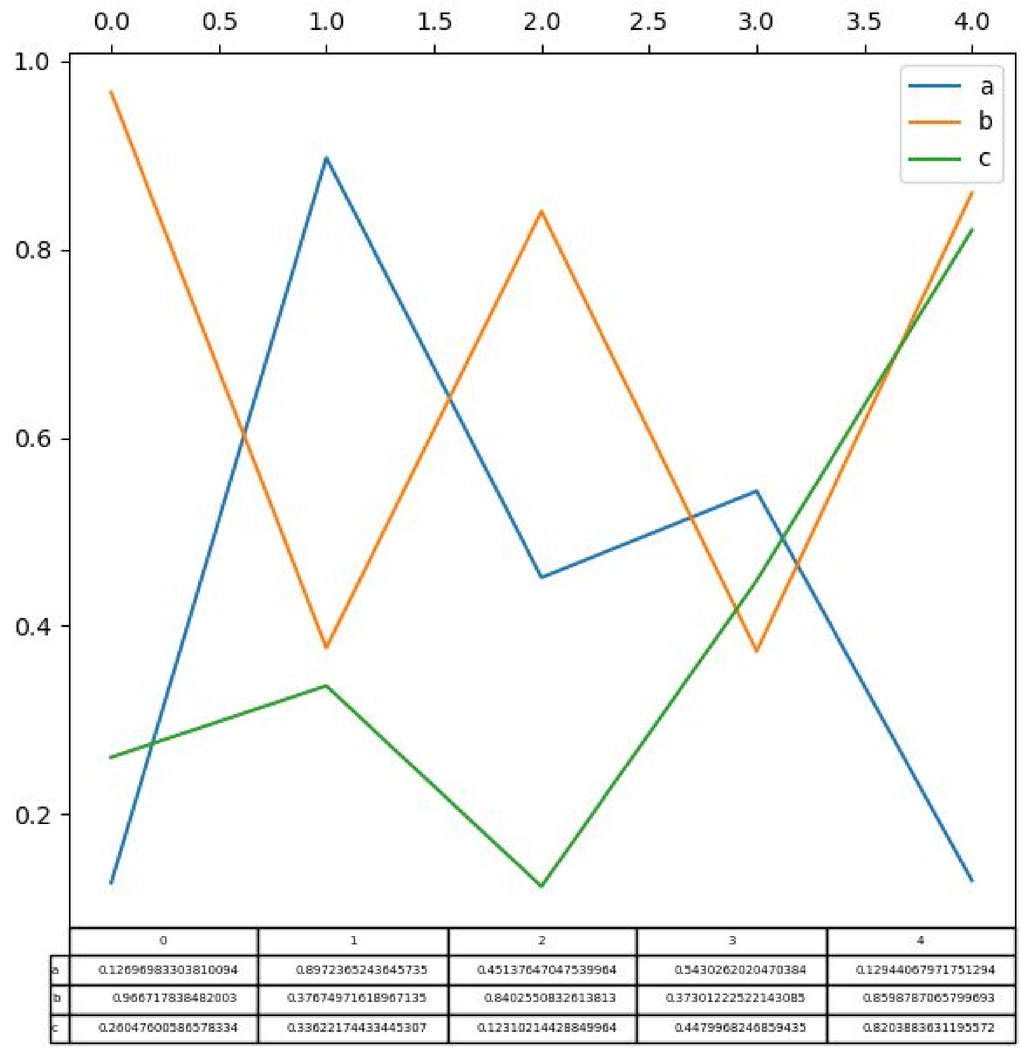
Plotting tables

Plotting with matplotlib table is now supported in DataFrame.plot() and Series.plot() with a table keyword. The table keyword can accept bool, DataFrame or Series. The simple way to draw a table is to specify table=True. Data will be transposed to meet matplotlib's default layout.

```
In [178]: fig, ax = plt.subplots(1, 1, figsize=(7, 6.5))
In [179]: df = pd.DataFrame(np.random.rand(5, 3), columns=["a", "b", "c"])
```

```
In [180]: ax.xaxis.tick_top()  # Display x-axis ticks on top.
```

```
In [181]: df.plot(table=True, ax=ax);
```

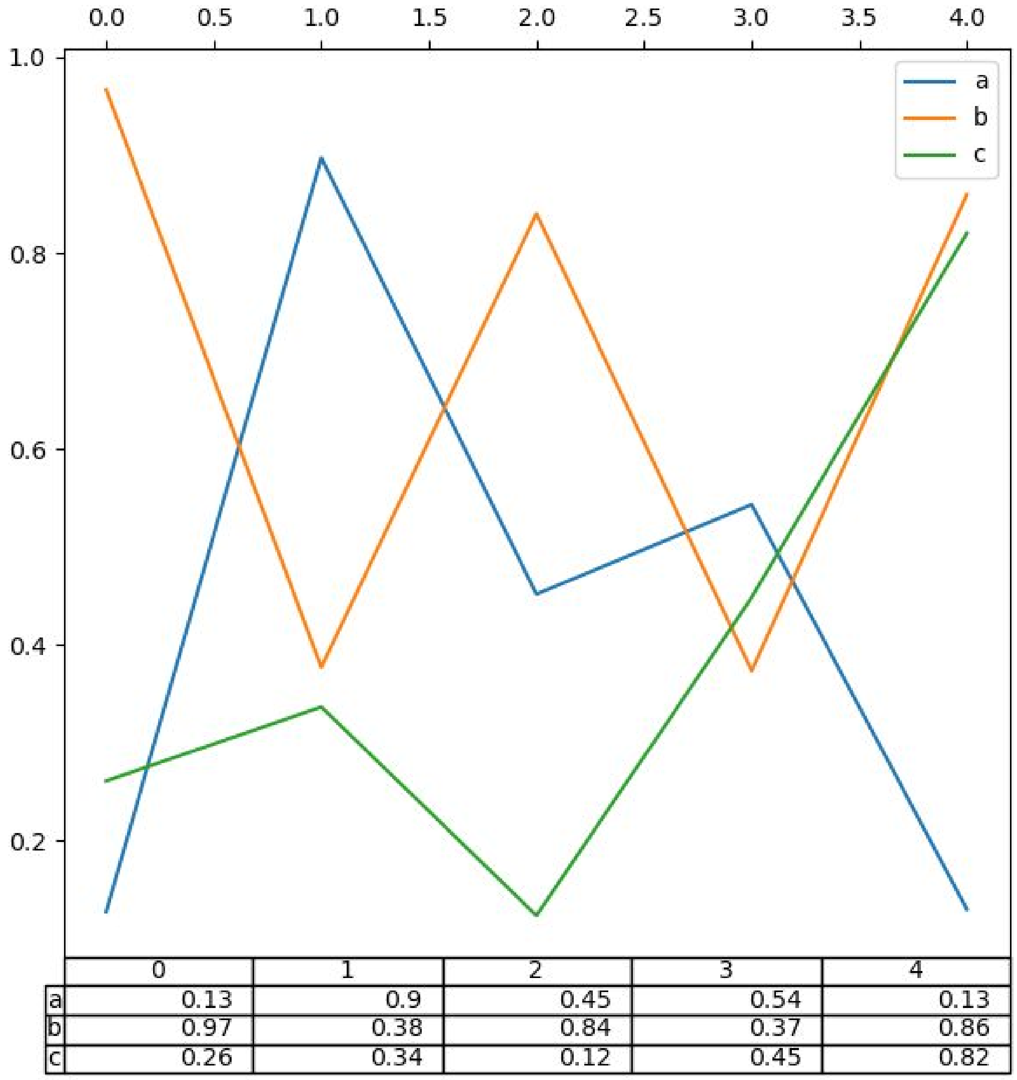


	0	1	2	3	4
a	0.12696983303810094	0.8972365243645735	0.45137647047539964	0.5430262020470384	0.12944067971751294
b	0.966717838482003	0.37674971618967135	0.8402550832613813	0.37301222522143085	0.8598787065799693
c	0.26047600586578334	0.33622174433445307	0.12310214428849964	0.44799682468859435	0.8203883631195572

Also, you can pass a different DataFrame or Series to the table keyword. The data will be drawn as displayed in print method (not transposed automatically). If required, it should be transposed manually as seen in the example below.

```
In [182]: fig, ax = plt.subplots(1, 1, figsize=(7, 6.75))
In [183]: ax.xaxis.tick_top()  # Display x-axis ticks on top.
```

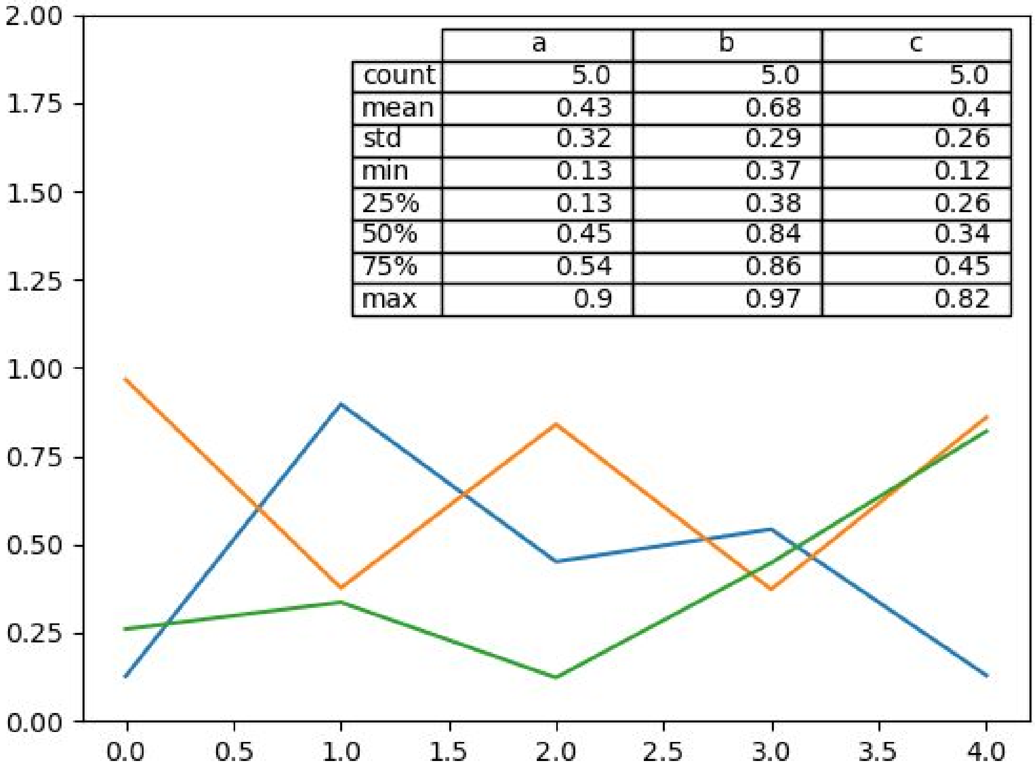
```
In [184]: df.plot(table=np.round(df.T, 2), ax=ax);
```



	0	1	2	3	4
a	0.13	0.9	0.45	0.54	0.13
b	0.97	0.38	0.84	0.37	0.86
c	0.26	0.34	0.12	0.45	0.82

There also exists a helper function pandas.plotting.table, which creates a table from DataFrame or Series, and adds it to an matplotlib.Axes instance. This function can accept keywords which the matplotlib table has.

```
In [185]: from pandas.plotting import tabl
In [186]: fig, ax = plt.subplots(1, 1)

In [187]: table(ax, np.round(df.describe(), 2), loc="upper right", colWidths=
[0.2, 0.2, 0.2]);

In [188]: df.plot(ax=ax, ylim=(0, 2), legend=None);
```



	a	b	c
count	5.0	5.0	5.0
mean	0.43	0.68	0.4
std	0.32	0.29	0.26
min	0.13	0.37	0.12
25%	0.13	0.38	0.26
50%	0.45	0.84	0.34
75%	0.54	0.86	0.45
max	0.9	0.97	0.82

Note: You can get table instances on the axes using axes.tables property for further decorations. See the matplotlib table documentation for more.

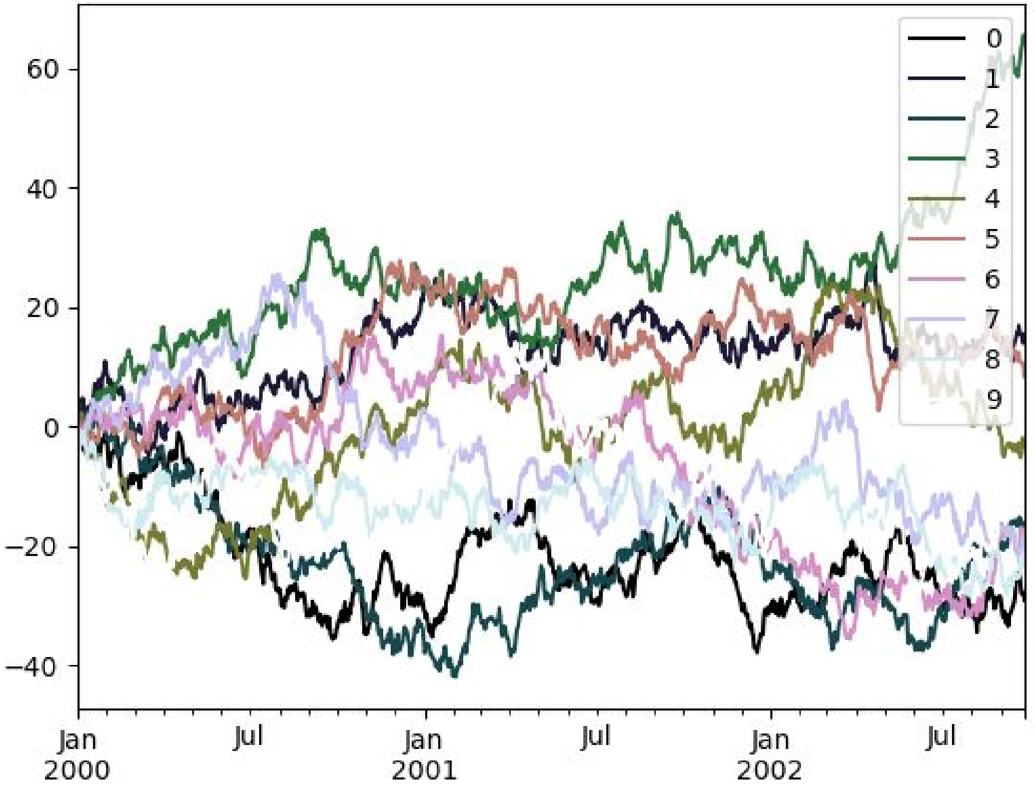
Colormaps

A potential issue when plotting a large number of columns is that it can be difficult to distinguish some series due to repetition in the default colors. To remedy this, DataFrame plotting supports the use of the colormap argument, which accepts either a Matplotlib colormap or a string that is a name of a colormap registered with Matplotlib. A visualization of the default matplotlib colormaps is available here.

As matplotlib does not directly support colormaps for line-based plots, the colors are selected based on an even spacing determined by the number of columns in the DataFrame. There is no consideration made for background color, so some colormaps will produce lines that are not easily visible.

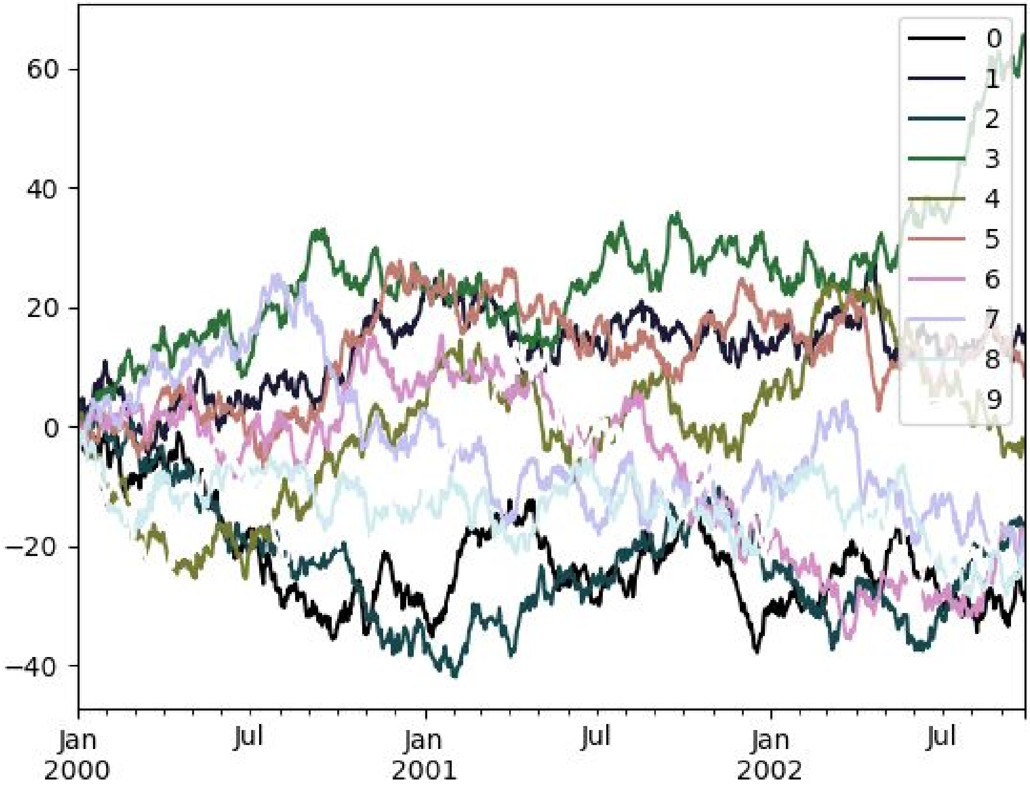
To use the cubehelix colormap, we can pass colormap='cubehelix'.

```
In [189]: df = pd.DataFrame(np.random.randn(1000, 10), index=ts.index)

In [190]: df = df.cumsum()

In [191]: plt.figure();

In [192]: df.plot(colormap="cubehelix");
```



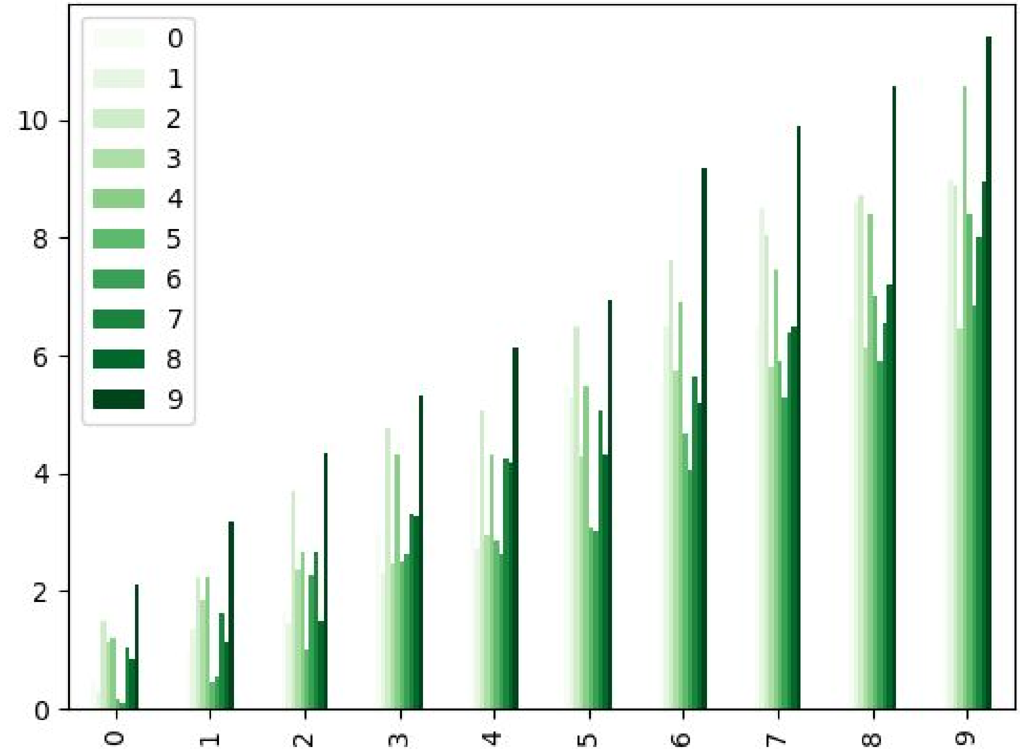
Alternatively, we can pass the colormap itself:

```
In [193]: from matplotlib import cm

In [194]: plt.figure();

In [195]: df.plot(colormap=cm.cubehelix);
```



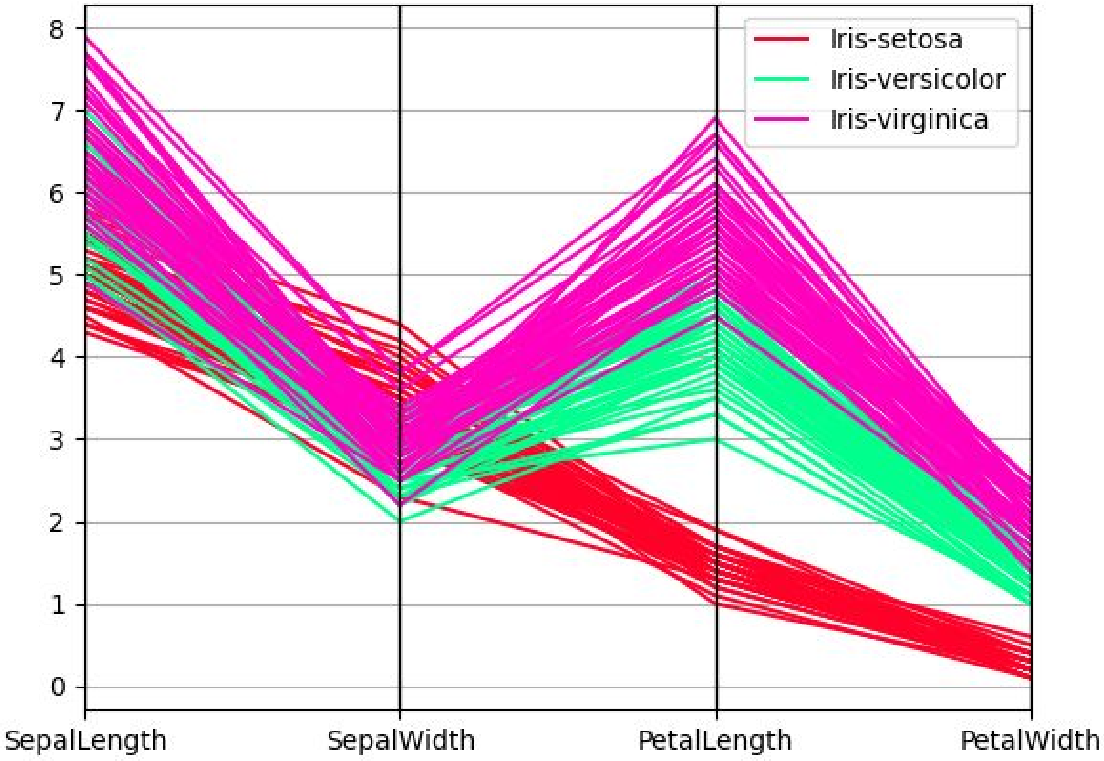
Colormaps can also be used other plot types, like bar charts:

```
In [196]: dd = pd.DataFrame(np.random.randn(10, 10)).applymap(abs)

In [197]: dd = dd.cumsum()

In [198]: plt.figure();

In [199]: dd.plot.bar(colormap="Greens");
```



Parallel coordinates charts:

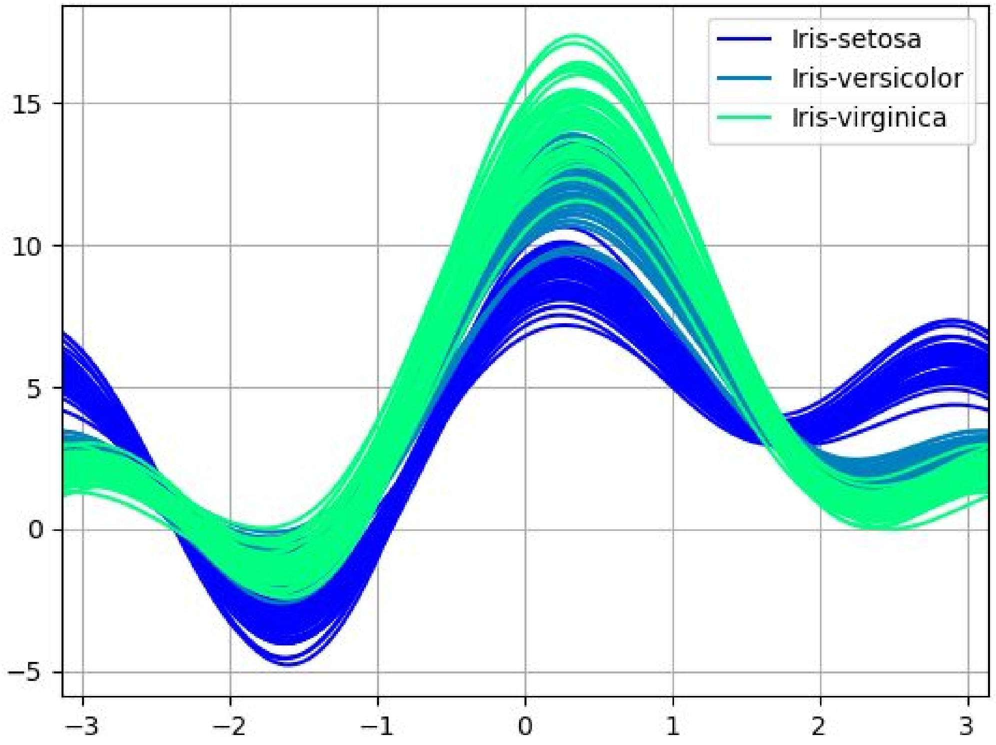
```
In [200]: plt.figure();
In [201]: parallel_coordinates(data, "Name", colormap="gist_rainbow");
```



Andrews curves charts:

```
In [202]: plt.figure();
```

```
In [203]: andrews_curves(data, "Name", colormap="winter");
```

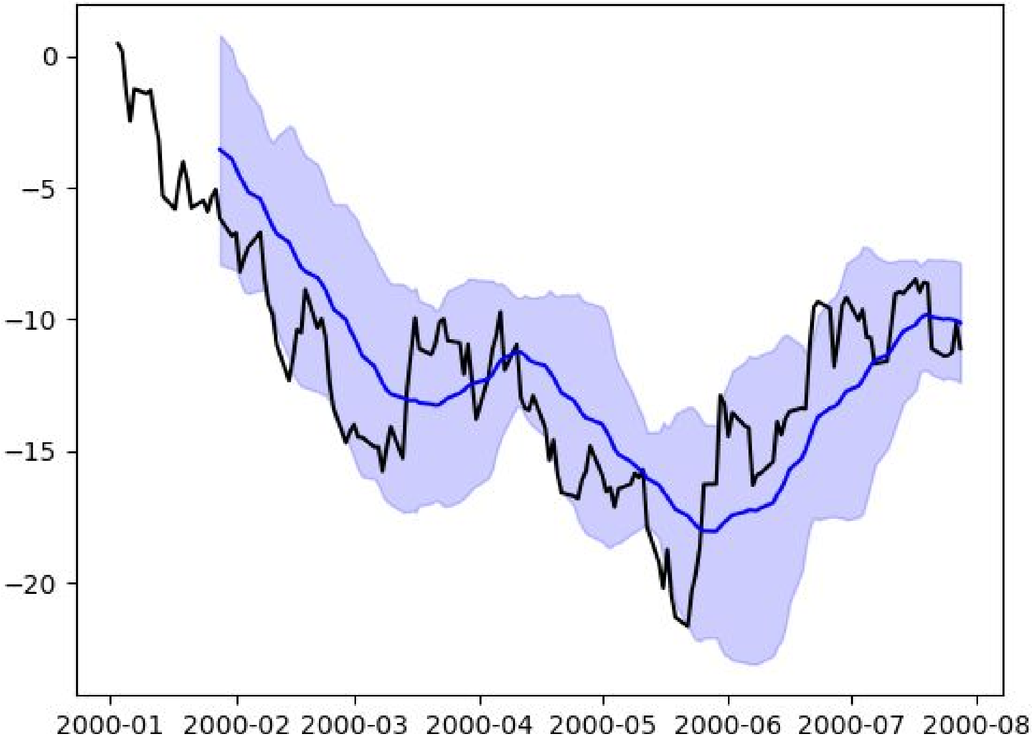


Plotting directly with matplotlib

In some situations it may still be preferable or necessary to prepare plots directly with matplotlib, for instance when a certain type of plot or customization is not (yet) supported by pandas. Series and DataFrame objects behave like arrays and can therefore be passed directly to matplotlib functions without explicit casts.

pandas also automatically registers formatters and locators that recognize date indices, thereby extending date and time support to practically all plot types available in matplotlib. Although this formatting does not provide the same level of refinement you would get when plotting via pandas, it can be faster when plotting a large number of points.

```
In [204]: price = pd.Series(
   .....:    np.random.randn(150).cumsum(),
   .....:    index=pd.date_range("2000-1-1", periods=150, freq="B"),
   .....: )
   .....:

In [205]: ma = price.rolling(20).mean()

In [206]: mstd = price.rolling(20).std()

In [207]: plt.figure();

In [208]: plt.plot(price.index, price, "k");

In [209]: plt.plot(ma.index, ma, "b");

In [210]: plt.fill_between(mstd.index, ma - 2 * mstd, ma + 2 * mstd, color="b",
alpha=0.2);
```



Plotting backends

Starting in version 0.25, pandas can be extended with third-party plotting backends. The main idea is letting users select a plotting backend different than the provided one based on Matplotlib.

This can be done by passing 'backend.module' as the argument backend in plot function. For example:

```
>>> Series([1, 2, 3]).plot(backend="backend.module")
```

Alternatively, you can also set this option globally, do you don't need to specify the keyword in each plot call. For example:

```
>>> pd.set_option("plotting.backend", "backend.module")
>>> pd.Series([1, 2, 3]).plot()
```

Or:

```
>>> pd.options.plotting.backend = "backend.module"
>>> pd.Series([1, 2, 3]).plot()
```

This would be more or less equivalent to:

```
>>> import backend.module
>>> backend.module.plot(pd.Series([1, 2, 3]))
```

The backend module can then use other visualization tools (Bokeh, Altair, hvplot,…) to generate the plots. Some libraries implementing a backend for pandas are listed on the ecosystem Visualization page.

Scaling to large datasets

pandas provides data structures for in-memory analytics, which makes using pandas to analyze datasets that are larger than memory datasets somewhat tricky. Even datasets that are a sizable fraction of memory become unwieldy, as some pandas operations need to make intermediate copies.

This document provides a few recommendations for scaling your analysis to larger datasets. It’s a complement to Enhancing performance, which focuses on speeding up analysis for datasets that fit in memory.

But first, it's worth considering *not using pandas*. pandas isn't the right tool for all situations. If you're working with very large datasets and a tool like PostgreSQL fits your needs, then you should probably be using that. Assuming you want or need the expressiveness and power of pandas, let's carry on.

```
In [1]: import pandas as pd

In [2]: import numpy as np
```

Load less data

Suppose our raw dataset on disk has many columns:

```
                id_0    name_0      x_0      y_0  id_1   name_1      x_1  ...
name_8      x_8      y_8  id_9   name_9      x_9      y_9

timestamp                                                                ...

2000-01-01 00:00:00  1015   Michael -0.399453  0.095427   994    Frank
-0.176842  ...     Dan -0.315310  0.713892  1025   Victor -0.135779  0.346801

2000-01-01 00:01:00   969  Patricia  0.650773 -0.874275  1003    Laura
0.459153  ...  Ursula  0.913244 -0.630308  1047    Wendy -0.886285
0.035852

2000-01-01 00:02:00  1016    Victor -0.721465 -0.584710  1046  Michael
0.524994  ...     Ray -0.656593  0.692568  1064   Yvonne  0.070426  0.432047

2000-01-01 00:03:00   939     Alice -0.746004 -0.908008   996   Ingrid
-0.414523  ...   Jerry -0.958994  0.608210   978    Wendy  0.855949
-0.648988

2000-01-01 00:04:00  1017       Dan  0.919451 -0.803504  1048    Jerry
-0.569235  ...   Frank -0.577022 -0.409088   994      Bob -0.270132  0.335176

...                   ...       ...       ...       ...   ...      ...       ...  ...      ...       ...      ...   ...
...       ...       ...
```

```
2000-12-30 23:56:00   999      Tim  0.162578  0.512817   973    Kevin
-0.403352  ...     Tim -0.380415  0.008097  1041  Charlie  0.191477
-0.599519

2000-12-30 23:57:00   970     Laura -0.433586 -0.600289   958   Oliver
-0.966577  ...   Zelda  0.971274  0.402032  1038   Ursula  0.574016
-0.930992

2000-12-30 23:58:00  1065     Edith  0.232211 -0.454540   971      Tim
0.158484  ...   Alice -0.222079 -0.919274  1022      Dan  0.031345 -0.657755

2000-12-30 23:59:00  1019    Ingrid  0.322208 -0.615974   981   Hannah
0.607517  ...   Sarah -0.424440 -0.117274   990   George -0.375530  0.563312

2000-12-31 00:00:00   937    Ursula -0.906523  0.943178  1018    Alice
-0.564513  ...   Jerry  0.236837  0.807650   985   Oliver  0.777642  0.783392


[525601 rows x 40 columns]
```

To load the columns we want, we have two options. Option 1 loads in all the data and then filters to what we need.

```
In [3]: columns = ["id_0", "name_0", "x_0", "y_0"]

In [4]: pd.read_parquet("timeseries_wide.parquet")[columns]

Out[4]:

                     id_0    name_0       x_0       y_0

timestamp

2000-01-01 00:00:00  1015   Michael -0.399453  0.095427

2000-01-01 00:01:00   969  Patricia  0.650773 -0.874275

2000-01-01 00:02:00  1016    Victor -0.721465 -0.584710
```

```
2000-01-01 00:03:00   939     Alice -0.746004 -0.908008
2000-01-01 00:04:00  1017       Dan  0.919451 -0.803504
...                   ...       ...       ...       ...
2000-12-30 23:56:00   999       Tim  0.162578  0.512817
2000-12-30 23:57:00   970     Laura -0.433586 -0.600289
2000-12-30 23:58:00  1065     Edith  0.232211 -0.454540
2000-12-30 23:59:00  1019    Ingrid  0.322208 -0.615974
2000-12-31 00:00:00   937    Ursula -0.906523  0.943178

[525601 rows x 4 columns]
```

Option 2 only loads the columns we request.

```
In [5]: pd.read_parquet("timeseries_wide.parquet", columns=columns)
Out[5]:
                     id_0    name_0       x_0       y_0
timestamp
2000-01-01 00:00:00  1015   Michael -0.399453  0.095427
2000-01-01 00:01:00   969  Patricia  0.650773 -0.874275
2000-01-01 00:02:00  1016    Victor -0.721465 -0.584710
2000-01-01 00:03:00   939     Alice -0.746004 -0.908008
2000-01-01 00:04:00  1017       Dan  0.919451 -0.803504
...                   ...       ...       ...       ...
```

```
2000-12-30 23:56:00   999      Tim  0.162578  0.512817
2000-12-30 23:57:00   970    Laura -0.433586 -0.600289
2000-12-30 23:58:00  1065    Edith  0.232211 -0.454540
2000-12-30 23:59:00  1019   Ingrid  0.322208 -0.615974
2000-12-31 00:00:00   937   Ursula -0.906523  0.943178

[525601 rows x 4 columns]
```

If we were to measure the memory usage of the two calls, we’d see that specifying columns uses about 1/10th the memory in this case.

With pandas.read_csv(), you can specify usecols to limit the columns read into memory. Not all file formats that can be read by pandas provide an option to read a subset of columns.

Use efficient datatypes

The default pandas data types are not the most memory efficient. This is especially true for text data columns with relatively few unique values (commonly referred to as “low-cardinality” data). By using more efficient data types, you can store larger datasets in memory.

```
In [6]: ts = pd.read_parquet("timeseries.parquet")

In [7]: ts

Out[7]:

                       id     name         x         y
timestamp
2000-01-01 00:00:00  1029   Michael  0.278837  0.247932
```

```
2000-01-01 00:00:30  1010  Patricia  0.077144  0.490260
2000-01-01 00:01:00  1001    Victor  0.214525  0.258635
2000-01-01 00:01:30  1018     Alice -0.646866  0.822104
2000-01-01 00:02:00   991       Dan  0.902389  0.466665
...                   ...       ...       ...       ...
2000-12-30 23:58:00   992     Sarah  0.721155  0.944118
2000-12-30 23:58:30  1007    Ursula  0.409277  0.133227
2000-12-30 23:59:00  1009    Hannah -0.452802  0.184318
2000-12-30 23:59:30   978     Kevin -0.904728 -0.179146
2000-12-31 00:00:00   973    Ingrid -0.370763 -0.794667

[1051201 rows x 4 columns]
```

Now, let’s inspect the data types and memory usage to see where we should focus our attention.

```
In [8]: ts.dtypes
Out[8]:
id        int64
name     object
x       float64
y       float64
dtype: object
```

```
In [9]: ts.memory_usage(deep=True)  # memory usage in bytes

Out[9]:

Index     8409608

id        8409608

name     65537768

x         8409608

y         8409608

dtype: int64
```

The name column is taking up much more memory than any other. It has just a few unique values, so it’s a good candidate for converting to a Categorical. With a Categorical, we store each unique name once and use space-efficient integers to know which specific name is used in each row.

```
In [10]: ts2 = ts.copy()

In [11]: ts2["name"] = ts2["name"].astype("category")

In [12]: ts2.memory_usage(deep=True)

Out[12]:

Index    8409608

id       8409608

name     1053894

x        8409608

y        8409608
```

```
dtype: int64
```

We can go a bit further and downcast the numeric columns to their smallest types using pandas.to_numeric().

```
In [13]: ts2["id"] = pd.to_numeric(ts2["id"], downcast="unsigned")

In [14]: ts2[["x", "y"]] = ts2[["x", "y"]].apply(pd.to_numeric, downcast="float")

In [15]: ts2.dtypes
Out[15]:
id        uint16
name    category
x        float32
y        float32
dtype: object
```

```
In [16]: ts2.memory_usage(deep=True)
Out[16]:
Index    8409608
id       2102402
name     1053894
x        4204804
y        4204804
dtype: int64
```

```
In [17]: reduction = ts2.memory_usage(deep=True).sum() / ts.memory_usage(deep=True).sum()

In [18]: print(f"{reduction:0.2f}")

0.20
```

In all, we’ve reduced the in-memory footprint of this dataset to 1/5 of its original size.

See Categorical data for more on Categorical and dtypes for an overview of all of pandas’ dtypes.

Use chunking

Some workloads can be achieved with chunking: splitting a large problem like “convert this directory of CSVs to parquet” into a bunch of small problems (“convert this individual CSV file into a Parquet file. Now repeat that for each file in this directory.”). As long as each chunk fits in memory, you can work with datasets that are much larger than memory.

Note

Chunking works well when the operation you’re performing requires zero or minimal coordination between chunks. For more complicated workflows, you’re better off using another library.

Suppose we have an even larger “logical dataset” on disk that’s a directory of parquet files. Each file in the directory represents a different year of the entire dataset.

```
data

└── timeseries
```

```
├── ts-00.parquet
├── ts-01.parquet
├── ts-02.parquet
├── ts-03.parquet
├── ts-04.parquet
├── ts-05.parquet
├── ts-06.parquet
├── ts-07.parquet
├── ts-08.parquet
├── ts-09.parquet
├── ts-10.parquet
└── ts-11.parquet
```

Now we’ll implement an out-of-core value_counts. The peak memory usage of this workflow is the single largest chunk, plus a small series storing the unique value counts up to this point. As long as each individual file fits in memory, this will work for arbitrary-sized datasets.

```
In [19]: %%time
   ....: files = pathlib.Path("data/timeseries/").glob("ts*.parquet")
   ....: counts = pd.Series(dtype=int)
   ....: for path in files:
   ....:     df = pd.read_parquet(path)
   ....:     counts = counts.add(df["name"].value_counts(), fill_value=0)
   ....: counts.astype(int)
   ....:
CPU times: user 622 ms, sys: 110 ms, total: 733 ms
Wall time: 684 ms
Out[19]:
Alice      229802
Bob        229211
Charlie    229303
Dan        230621
Edith      230349
            ...
Victor     230502
Wendy      230038
Xavier     229553
Yvonne     228766
Zelda      229909
Length: 26, dtype: int64
```

Some readers, like pandas.read_csv(), offer parameters to control the chunksize when reading a single file.

Manually chunking is an OK option for workflows that don't require too sophisticated of operations. Some operations, like groupby, are much harder to do chunkwise. In these cases, you may be better switching to a different library that implements these out-of-core algorithms for you.

Use other libraries

pandas is just one library offering a DataFrame API. Because of its popularity, pandas' API has become something of a standard that other libraries implement. The pandas documentation maintains a list of libraries implementing a DataFrame API in our ecosystem page.

For example, Dask, a parallel computing library, has dask.dataframe, a pandas-like API for working with larger than memory datasets in parallel. Dask can use multiple threads or processes on a single machine, or a cluster of machines to process data in parallel.

We'll import dask.dataframe and notice that the API feels similar to pandas. We can use Dask's read_parquet function, but provide a globstring of files to read in.

```
In [20]: import dask.dataframe as dd
```

```
In [21]: ddf = dd.read_parquet("data/timeseries/ts*.parquet", engine="pyarrow")
```

```
In [22]: ddf
```

```
Out[22]:

Dask DataFrame Structure:

                id   name      x       y

npartitions=12

             int64  object  float64  float64
```

```
               ...    ...    ...    ...
...            ...    ...    ...    ...
               ...    ...    ...    ...
               ...    ...    ...    ...
Dask Name: read-parquet, 12 tasks
```

Inspecting the ddf object, we see a few things

- There are familiar attributes like .columns and .dtypes
- There are familiar methods like .groupby, .sum, etc.
- There are new attributes like .npartitions and .divisions

The partitions and divisions are how Dask parallelizes computation. A Dask DataFrame is made up of many pandas DataFrames. A single method call on a Dask DataFrame ends up making many pandas method calls, and Dask knows how to coordinate everything to get the result.

```
In [23]: ddf.columns

Out[23]: Index(['id', 'name', 'x', 'y'], dtype='object')


In [24]: ddf.dtypes

Out[24]:

id       int64

name     object

x       float64

y       float64

dtype: object
```

```
In [25]: ddf.npartitions
```

```
Out[25]: 12
```

One major difference: the dask.dataframe API is *lazy*. If you look at the repr above, you'll notice that the values aren't actually printed out; just the column names and dtypes. That's because Dask hasn't actually read the data yet. Rather than executing immediately, doing operations build up a task graph.

```
In [26]: ddf
```

```
Out[26]:

Dask DataFrame Structure:

                id   name      x       y

npartitions=12

             int64  object  float64  float64

               ...    ...      ...      ...

...            ...    ...      ...      ...

               ...    ...      ...      ...

               ...    ...      ...      ...

Dask Name: read-parquet, 12 tasks
```

```
In [27]: ddf["name"]
```

```
Out[27]:

Dask Series Structure:

npartitions=12
```

```
    object
       ...
     ...
       ...
       ...
Name: name, dtype: object
Dask Name: getitem, 24 tasks
```

```
In [28]: ddf["name"].value_counts()
Out[28]:
Dask Series Structure:
npartitions=1
    int64
      ...
Name: name, dtype: int64
Dask Name: value-counts-agg, 39 tasks
```

Each of these calls is instant because the result isn’t being computed yet. We’re just building up a list of computation to do when someone needs the result. Dask knows that the return type of a pandas.Series.value_counts is a pandas Series with a certain dtype and a certain name. So the Dask version returns a Dask Series with the same dtype and the same name.

To get the actual result you can call .compute().

```
In [29]: %time ddf["name"].value_counts().compute()
CPU times: user 885 ms, sys: 77.5 ms, total: 962 ms
Wall time: 526 ms
Out[29]:
Laura      230906
Ingrid     230838
Kevin      230698
Dan        230621
Frank      230595
            ...
Ray        229603
Xavier     229553
Charlie    229303
Bob        229211
Yvonne     228766
Name: name, Length: 26, dtype: int64
```

At that point, you get back the same thing you'd get with pandas, in this case a concrete pandas Series with the count of each name.

Calling .compute causes the full task graph to be executed. This includes reading the data, selecting the columns, and doing the value_counts. The execution is done *in parallel* where possible, and Dask tries to keep the overall memory footprint small. You can work with datasets that are much larger than memory, as long as each partition (a regular pandas DataFrame) fits in memory.

By default, dask.dataframe operations use a threadpool to do operations in parallel. We can also connect to a cluster to distribute the work on many machines. In this case we’ll connect to a local “cluster” made up of several processes on this single machine.

```
>>> from dask.distributed import Client, LocalCluster

>>> cluster = LocalCluster()

>>> client = Client(cluster)

>>> client

<Client: 'tcp://127.0.0.1:53349' processes=4 threads=8, memory=17.18 GB>
```

Once this client is created, all of Dask’s computation will take place on the cluster (which is just processes in this case).

Dask implements the most used parts of the pandas API. For example, we can do a familiar groupby aggregation.

```
In [30]: %time ddf.groupby("name")[["x", "y"]].mean().compute().head()

CPU times: user 1.88 s, sys: 302 ms, total: 2.18 s

Wall time: 899 ms

Out[30]:

              x        y

name

Alice    0.000086 -0.001170

Bob     -0.000843 -0.000799
```

```
Charlie  0.000564 -0.000038
Dan      0.000584  0.000818
Edith   -0.000116 -0.000044
```

The grouping and aggregation is done out-of-core and in parallel.

When Dask knows the divisions of a dataset, certain optimizations are possible. When reading parquet datasets written by dask, the divisions will be known automatically. In this case, since we created the parquet files manually, we need to supply the divisions manually.

```
In [31]: N = 12

In [32]: starts = [f"20{i:>02d}-01-01" for i in range(N)]

In [33]: ends = [f"20{i:>02d}-12-13" for i in range(N)]

In [34]: divisions = tuple(pd.to_datetime(starts)) + (pd.Timestamp(ends[-1]),)


In [35]: ddf.divisions = divisions


In [36]: ddf
Out[36]:
Dask DataFrame Structure:
                   id    name       x       y
npartitions=12
2000-01-01      int64  object  float64  float64
2001-01-01        ...     ...      ...      ...
```

```
...              ...    ...     ...     ...
2011-01-01        ...     ...     ...     ...
2011-12-13        ...     ...     ...     ...
Dask Name: read-parquet, 12 tasks
```

Now we can do things like fast random access with .loc.

```
In [37]: ddf.loc["2002-01-01 12:01":"2002-01-01 12:05"].compute()
```

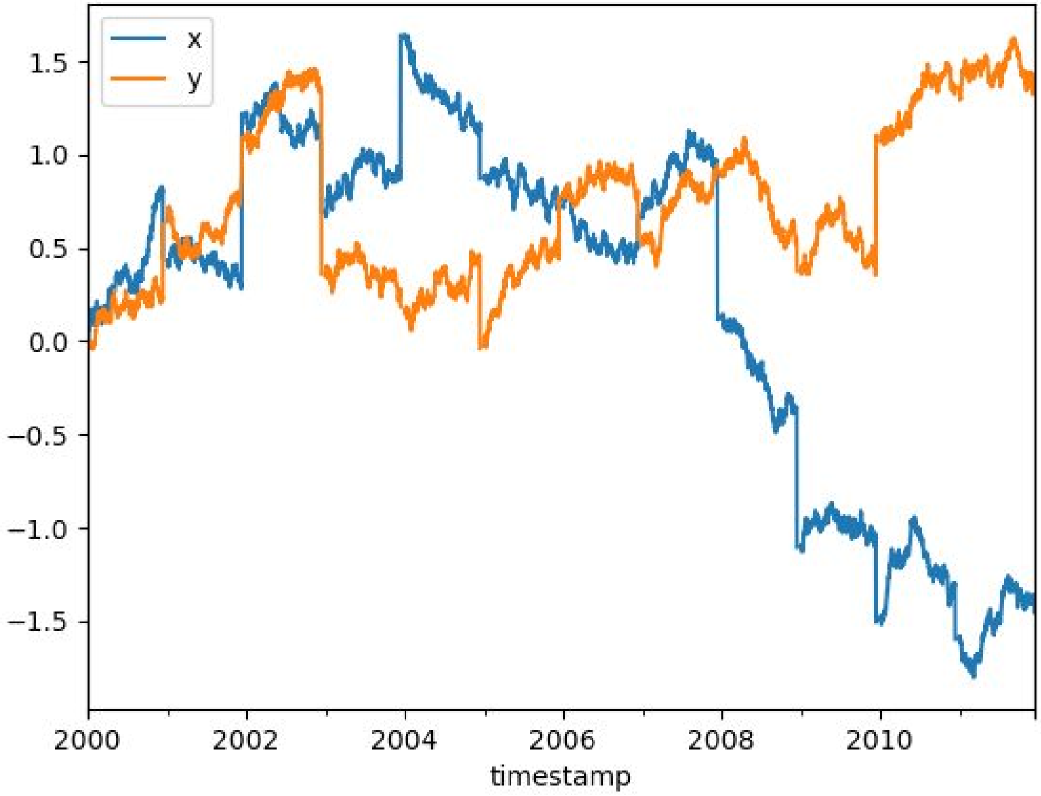
```
Out[37]:
                       id    name         x         y
timestamp
2002-01-01 12:01:00   983   Laura  0.243985 -0.079392
2002-01-01 12:02:00  1001   Laura -0.523119 -0.226026
2002-01-01 12:03:00  1059  Oliver  0.612886  0.405680
2002-01-01 12:04:00   993   Kevin  0.451977  0.332947
2002-01-01 12:05:00  1014  Yvonne -0.948681  0.361748
```

Dask knows to just look in the 3rd partition for selecting values in 2002. It doesn't need to look at any other data.

Many workflows involve a large amount of data and processing it in a way that reduces the size to something that fits in memory. In this case, we'll resample to daily frequency and take the mean. Once we've taken the mean, we know the results will fit in memory, so we can safely call compute without running out of memory. At that point it's just a regular pandas object.

```
In [38]: ddf[["x", "y"]].resample("1D").mean().cumsum().compute().plot()
```

```
Out[38]: <AxesSubplot:xlabel='timestamp'>
```



These Dask examples have all be done using multiple processes on a single machine. Dask can be deployed on a cluster to scale up to even larger datasets.

Enhancing performance

In this part of the tutorial, we will investigate how to speed up certain functions operating on pandas DataFrames using three different techniques: Cython, Numba and pandas.eval(). We will see a speed improvement of ~200 when we use Cython and Numba on a test function operating row-wise on the DataFrame. Using pandas.eval() we will speed up a sum by an order of ~2.

Note

In addition to following the steps in this tutorial, users interested in enhancing performance are highly encouraged to install the recommended dependencies for pandas. These dependencies are often not installed by default, but will offer speed improvements if present.

Cython (writing C extensions for pandas)

For many use cases writing pandas in pure Python and NumPy is sufficient. In some computationally heavy applications however, it can be possible to achieve sizable speed-ups by offloading work to cython.

This tutorial assumes you have refactored as much as possible in Python, for example by trying to remove for-loops and making use of NumPy vectorization. It's always worth optimising in Python first.

This tutorial walks through a “typical” process of cythonizing a slow computation. We use an example from the Cython documentation but in the context of pandas. Our final cythonized solution is around 100 times faster than the pure Python solution.

Pure Python

We have a DataFrame to which we want to apply a function row-wise.

```
In [1]: df = pd.DataFrame(
   ...:     {
   ...:         "a": np.random.randn(1000),
   ...:         "b": np.random.randn(1000),
   ...:         "N": np.random.randint(100, 1000, (1000)),
   ...:         "x": "x",
   ...:     }
   ...: )
   ...:

In [2]: df
Out[2]:
          a         b    N  x
0    0.469112 -0.218470  585  x
1   -0.282863 -0.061645  841  x
2   -1.509059 -0.723780  251  x
3   -1.135632  0.551225  972  x
4    1.212112 -0.497767  181  x
..        ...       ...  ... ..
995 -1.512743  0.874737  374  x
996  0.933753  1.120790  246  x
997 -0.308013  0.198768  157  x
998 -0.079915  1.757555  977  x
999 -1.010589 -1.115680  770  x

[1000 rows x 4 columns]
```

Here’s the function in pure Python:

```
In [3]: def f(x):
   ...:     return x * (x - 1)
   ...:


In [4]: def integrate_f(a, b, N):
   ...:     s = 0
   ...:     dx = (b - a) / N
   ...:     for i in range(N):
   ...:         s += f(a + i * dx)
   ...:     return s * dx
   ...:
```

We achieve our result by using apply (row-wise):

```
In [7]: %timeit df.apply(lambda x: integrate_f(x["a"], x["b"], x["N"]), axis=1)
10 loops, best of 3: 174 ms per loop
```

But clearly this isn't fast enough for us. Let's take a look and see where the time is spent during this operation (limited to the most time consuming four calls) using the prun ipython magic function:

```
In [5]: %prun -l 4 df.apply(lambda x: integrate_f(x["a"], x["b"], x["N"]), axis=1)  # noqa E999
         638348 function calls (638330 primitive calls) in 0.309 seconds


   Ordered by: internal time
```

```
  List reduced from 228 to 4 due to restriction <4>

  ncalls  tottime  percall  cumtime  percall filename:lineno(function)
    1000    0.155    0.000    0.225    0.000 <ipython-input-4-
c2a74e076cf0>:1(integrate_f)
  552423    0.070    0.000    0.070    0.000 <ipython-input-3-c138bd-
d570e3>:1(f)
    3000    0.009    0.000    0.047    0.000 series.py:943(__getitem__)
    3000    0.006    0.000    0.030    0.000 series.py:1052(_get_value)
```

By far the majority of time is spend inside either integrate_f or f, hence we'll concentrate our efforts cythonizing these two functions.

Plain Cython

First we're going to need to import the Cython magic function to IPython:

```
In [6]: %load_ext Cython
```

Now, let's simply copy our functions over to Cython as is (the suffix is here to distinguish between function versions):

```
In [7]: %%cython
   ...: def f_plain(x):
   ...:     return x * (x - 1)
   ...: def integrate_f_plain(a, b, N):
   ...:     s = 0
   ...:     dx = (b - a) / N
   ...:     for i in range(N):
   ...:         s += f_plain(a + i * dx)
   ...:     return s * dx
   ...:
```

Note

If you’re having trouble pasting the above into your ipython, you may need to be using bleeding edge IPython for paste to play well with cell magics.

```
In [4]: %timeit df.apply(lambda x: integrate_f_plain(x["a"], x["b"], x["N"]), axis=1)
10 loops, best of 3: 85.5 ms per loop
```

Already this has shaved a third off, not too bad for a simple copy and paste.

Adding type

We get another huge improvement simply by providing type information:

```
In [8]: %%cython
   ...: cdef double f_typed(double x) except? -2:
   ...:     return x * (x - 1)
   ...: cpdef double integrate_f_typed(double a, double b, int N):
   ...:     cdef int i
   ...:     cdef double s, dx
   ...:     s = 0
   ...:     dx = (b - a) / N
   ...:     for i in range(N):
   ...:         s += f_typed(a + i * dx)
   ...:     return s * dx
   ...:
```

```
In [4]: %timeit df.apply(lambda x: integrate_f_typed(x["a"], x["b"], x["N"]), axis=1)
10 loops, best of 3: 20.3 ms per loop
```

Now, we're talking! It's now over ten times faster than the original Python implementation, and we haven't *really* modified the code. Let's have another look at what's eating up time:

```
In [9]: %prun -l 4 df.apply(lambda x: integrate_f_typed(x["a"], x["b"], x["N"]), axis=1)
         85918 function calls (85900 primitive calls) in 0.052 seconds

   Ordered by: internal time
   List reduced from 221 to 4 due to restriction <4>

   ncalls  tottime  percall  cumtime  percall filename:lineno(function)
     3000    0.006    0.000    0.031    0.000 series.py:943(__getitem__)
     3000    0.004    0.000    0.018    0.000 series.py:1052(_get_value)
```

```
 16189    0.003    0.000    0.004    0.000 {built-in method builtins.isinstance}
  3000    0.003    0.000    0.005    0.000 base.py:5646(_get_values_for_loc)
```

Using ndarray

It’s calling series… a lot! It’s creating a Series from each row, and get-ting from both the index and the series (three times for each row). Function calls are expensive in Python, so maybe we could minimize these by cythonizing the apply part.

Note

We are now passing ndarrays into the Cython function, fortunately Cython plays very nicely with NumPy.

```
In [10]: %%cython

   ....: cimport numpy as np

   ....: import numpy as np

   ....: cdef double f_typed(double x) except? -2:

   ....:     return x * (x - 1)

   ....: cpdef double integrate_f_typed(double a, double b, int N):

   ....:     cdef int i

   ....:     cdef double s, dx

   ....:     s = 0

   ....:     dx = (b - a) / N

   ....:     for i in range(N):

   ....:         s += f_typed(a + i * dx)
```

```
....:     return s * dx
....: cpdef np.ndarray[double] apply_integrate_f(np.ndarray col_a, np.ndarray col_b,
....:                                            np.ndarray col_N):
....:     assert (col_a.dtype == np.float_
....:             and col_b.dtype == np.float_ and col_N.dtype == np.int_)
....:     cdef Py_ssize_t i, n = len(col_N)
....:     assert (len(col_a) == len(col_b) == n)
....:     cdef np.ndarray[double] res = np.empty(n)
....:     for i in range(len(col_a)):
....:         res[i] = integrate_f_typed(col_a[i], col_b[i], col_N[i])
....:     return res
....:
```

The implementation is simple, it creates an array of zeros and loops over the rows, applying our integrate_f_typed, and putting this in the zeros array.

Warning

You can not pass a Series directly as a ndarray typed parameter to a Cython function. Instead pass the actual ndarray using the Series.to_numpy(). The reason is that the Cython definition is specific to an ndarray and not the passed Series.

So, do not do this:

```
apply_integrate_f(df["a"], df["b"], df["N"])
```

But rather, use Series.to_numpy() to get the underlying ndarray:

```
apply_integrate_f(df["a"].to_numpy(), df["b"].to_numpy(),
df["N"].to_numpy())
```

Note

Loops like this would be *extremely* slow in Python, but in Cython looping over NumPy arrays is *fast*.

```
In [4]: %timeit apply_integrate_f(df["a"].to_numpy(), df["b"].to_numpy(),
df["N"].to_numpy())

1000 loops, best of 3: 1.25 ms per loop
```

We’ve gotten another big improvement. Let’s check again where the time is spent:

```
In [11]: %prun -l 4 apply_integrate_f(df["a"].to_numpy(), df["b"].to_numpy(),
df["N"].to_numpy())

         200 function calls in 0.002 seconds


   Ordered by: internal time

   List reduced from 53 to 4 due to restriction <4>


   ncalls  tottime  percall  cumtime  percall filename:lineno(function)
```

```
     1    0.001    0.001    0.001    0.001 {built-in method _cython_magic_ddc107c474f1147882a8a0afef7a24e2.apply_integrate_f}
     3    0.000    0.000    0.000    0.000 frame.py:3463(__getitem__)
     3    0.000    0.000    0.000    0.000 managers.py:1016(iget)
     1    0.000    0.000    0.002    0.002 {built-in method builtins.exec}
```

As one might expect, the majority of the time is now spent in apply_integrate_f, so if we wanted to make anymore efficiencies we must continue to concentrate our efforts here.

More advanced techniques

There is still hope for improvement. Here's an example of using some more advanced Cython techniques:

```
In [12]: %%cython
   ....: cimport cython
   ....: cimport numpy as np
   ....: import numpy as np
   ....: cdef double f_typed(double x) except? -2:
   ....:     return x * (x - 1)
   ....: cpdef double integrate_f_typed(double a, double b, int N):
   ....:     cdef int i
   ....:     cdef double s, dx
   ....:     s = 0
   ....:     dx = (b - a) / N
   ...:     for i in range(N):
   ...:         s += f_typed(a + i * dx)
   ...:     return s * dx
   ...: @cython.boundscheck(False)
   ...: @cython.wraparound(False)
   ...: cpdef np.ndarray[double] apply_integrate_f_wrap(np.ndarray[double] col_a,
   ...:                                                 np.ndarray[double] col_b,
   ...:                                                 np.ndarray[int] col_N):
   ...:     cdef int i, n = len(col_N)
   ...:     assert len(col_a) == len(col_b) == n
   ...:     cdef np.ndarray[double] res = np.empty(n)
   ...:     for i in range(n):
   ...:         res[i] = integrate_f_typed(col_a[i], col_b[i], col_N[i])
   ...:     return res
   ...:
```

```
In [4]: %timeit apply_integrate_f_wrap(df["a"].to_numpy(), df["b"].to_numpy(), df["N"].to_numpy())
1000 loops, best of 3: 987 us per loop
```

Even faster, with the caveat that a bug in our Cython code (an off-by-one error, for example) might cause a segfault because memory access isn’t checked. For more about boundscheck and wraparound, see the Cython docs on compiler directives.

Numba (JIT compilation)

An alternative to statically compiling Cython code is to use a dynamic just-in-time (JIT) compiler with Numba.

Numba allows you to write a pure Python function which can be JIT compiled to native machine instructions, similar in performance to C, C++ and Fortran, by decorating your function with @jit.

Numba works by generating optimized machine code using the LLVM compiler infrastructure at import time, runtime, or statically (using the included pycc tool). Numba supports compilation of Python to run on either CPU or GPU hardware and is designed to integrate with the Python scientific software stack.

Note

The @jit compilation will add overhead to the runtime of the function, so performance benefits may not be realized especially when using small data sets. Consider caching your function to avoid compilation overhead each time your function is run.

Numba can be used in 2 ways with pandas:

1. Specify the engine="numba" keyword in select pandas methods
2. Define your own Python function decorated with @jit and pass the underlying NumPy array of Series or Dataframe (using to_numpy()) into the function

pandas Numba Engine

If Numba is installed, one can specify engine="numba" in select pandas methods to execute the method using Numba. Methods that support engine="numba" will also have an engine_kwargs keyword that accepts a dictionary that allows one to specify "nogil", "nopython" and "parallel" keys with boolean values to pass into the @jit decorator. If engine_kwargs is not specified, it defaults to {"nogil": False, "nopython": True, "parallel": False} unless otherwise specified.

In terms of performance, the first time a function is run using the Numba engine will be slow as Numba will have some function compilation overhead. However, the JIT compiled functions are cached, and subsequent calls will be fast. In general, the Numba engine is performant with a larger amount of data points (e.g. 1+ million).

```
In [1]: data = pd.Series(range(1_000_000))  # noqa: E225

In [2]: roll = data.rolling(10)

In [3]: def f(x):

   ...:     return np.sum(x) + 5

# Run the first time, compilation time will affect performance

In [4]: %timeit -r 1 -n 1 roll.apply(f, engine='numba', raw=True)

1.23 s ± 0 ns per loop (mean ± std. dev. of 1 run, 1 loop each)

# Function is cached and performance will improve

In [5]: %timeit roll.apply(f, engine='numba', raw=True)

188 ms ± 1.93 ms per loop (mean ± std. dev. of 7 runs, 10 loops each)


In [6]: %timeit roll.apply(f, engine='cython', raw=True)

3.92 s ± 59 ms per loop (mean ± std. dev. of 7 runs, 1 loop each)
```

Caveats

Numba is best at accelerating functions that apply numerical functions to NumPy arrays. If you try to @jit a function that contains unsupported Python or NumPy code, compilation will revert object mode which will mostly likely not speed up your function. If you would prefer that Numba throw an error if it cannot compile a function in a way that speeds up your code, pass Numba the argument nopython=True (e.g. @jit(nopython=True)). For more on troubleshooting Numba modes, see the Numba troubleshooting page.

Using parallel=True (e.g. @jit(parallel=True)) may result in a SIGABRT if the threading layer leads to unsafe behavior. You can first specify a safe threading layer before running a JIT function with parallel=True.

Generally if the you encounter a segfault (SIGSEGV) while using Numba, please report the issue to the Numba issue tracker.

Expression evaluation via eval()

The top-level function pandas.eval() implements expression evaluation of Series and DataFrame objects.

Note

To benefit from using eval() you need to install numexpr. See the recommended dependencies section for more details.

The point of using eval() for expression evaluation rather than plain Python is two-fold: 1) large DataFrame objects are evaluated more efficiently and 2) large arithmetic and boolean expressions are evaluated all at once by the underlying engine (by default numexpr is used for evaluation).

Note

You should not use eval() for simple expressions or for expressions involving small DataFrames. In fact, eval() is many orders of magnitude slower for smaller expressions/objects than plain ol’ Python. A good rule of thumb is to only use eval() when you have a DataFrame with more than 10,000 rows.

eval() supports all arithmetic expressions supported by the engine in addition to some extensions available only in pandas.

Note

The larger the frame and the larger the expression the more speedup you will see from using eval().

Supported syntax

These operations are supported by pandas.eval():

- Arithmetic operations except for the left shift (<<) and right shift (>>) operators, e.g., df + 2 * pi / s ** 4 % 42 - the_golden_ratio
- Comparison operations, including chained comparisons, e.g., 2 < df < df2
- Boolean operations, e.g., df < df2 and df3 < df4 or not df_bool
- list and tuple literals, e.g., [1, 2] or (1, 2)
- Attribute access, e.g., df.a
- Subscript expressions, e.g., df[0]
- Simple variable evaluation, e.g., pd.eval("df") (this is not very useful)
- Math functions: sin, cos, exp, log, expm1, log1p, sqrt, sinh, cosh, tanh, arcsin, arccos, arctan, arccosh, arcsinh, arctanh, abs, arctan2 and log10.

This Python syntax is not allowed:

- Expressions
  - Function calls other than math functions.
  - is/is not operations
  - if expressions
  - lambda expressions
  - list/set/dict comprehensions
  - Literal dict and set expressions
  - yield expressions

  - Generator expressions
  - Boolean expressions consisting of only scalar values
- Statements
  - Neither simple nor compound statements are allowed. This includes things like for, while, and if.

eval() examples

pandas.eval() works well with expressions containing large arrays.

First let’s create a few decent-sized arrays to play with:

```
In [13]: nrows, ncols = 20000, 100

In [14]: df1, df2, df3, df4 = [pd.DataFrame(np.random.randn(nrows, ncols))
for _ in range(4)]
```

Now let’s compare adding them together using plain ol’ Python versus eval():

```
In [15]: %timeit df1 + df2 + df3 + df4

8.34 ms +- 147 us per loop (mean +- std. dev. of 7 runs, 100 loops each)
```

```
In [16]: %timeit pd.eval("df1 + df2 + df3 + df4")

5.71 ms +- 121 us per loop (mean +- std. dev. of 7 runs, 100 loops each)
```

Now let’s do the same thing but with comparisons:

```
In [17]: %timeit (df1 > 0) & (df2 > 0) & (df3 > 0) & (df4 > 0)

7.47 ms +- 199 us per loop (mean +- std. dev. of 7 runs, 100 loops each)
```

```
In [18]: %timeit pd.eval("(df1 > 0) & (df2 > 0) & (df3 > 0) & (df4 > 0)")

6.64 ms +- 62.7 us per loop (mean +- std. dev. of 7 runs, 100 loops each)
```

eval() also works with unaligned pandas objects:

```
In [19]: s = pd.Series(np.random.randn(50))


In [20]: %timeit df1 + df2 + df3 + df4 + s

23.8 ms +- 697 us per loop (mean +- std. dev. of 7 runs, 10 loops each)
```

```
In [21]: %timeit pd.eval("df1 + df2 + df3 + df4 + s")

6.3 ms +- 67.3 us per loop (mean +- std. dev. of 7 runs, 100 loops each)
```

Note

Operations such as

```
1 and 2  # would parse to 1 & 2, but should evaluate to 2

3 or 4  # would parse to 3 | 4, but should evaluate to 3

~1  # this is okay, but slower when using eval
```

should be performed in Python. An exception will be raised if you try to perform any boolean/bitwise operations with scalar operands that are not of type bool or np.bool_. Again, you should perform these kinds of operations in plain Python.

The DataFrame.eval method

In addition to the top level pandas.eval() function you can also evaluate an expression in the "context" of a DataFrame.

```
In [22]: df = pd.DataFrame(np.random.randn(5, 2), columns=["a", "b"])
```

```
In [23]: df.eval("a + b")
Out[23]:
0   -0.246747
1    0.867786
2   -1.626063
3   -1.134978
4   -1.027798
dtype: float64
```

Any expression that is a valid pandas.eval() expression is also a valid DataFrame.eval() expression, with the added benefit that you don't have to prefix the name of the DataFrame to the column(s) you're interested in evaluating.

In addition, you can perform assignment of columns within an expression. This allows for *formulaic evaluation*. The assignment target can be a new column name or an existing column name, and it must be a valid Python identifier.

The inplace keyword determines whether this assignment will performed on the original DataFrame or return a copy with the new column.

```
In [24]: df = pd.DataFrame(dict(a=range(5), b=range(5, 10)))
In [25]: df.eval("c = a + b", inplace=True)

In [26]: df.eval("d = a + b + c", inplace=True)

In [27]: df.eval("a = 1", inplace=True)

In [28]: df

Out[28]:

   a  b   c   d

0  1  5   5  10

1  1  6   7  14

2  1  7   9  18

3  1  8  11  22

4  1  9  13  26
```

When inplace is set to False, the default, a copy of the DataFrame with the new or modified columns is returned and the original frame is unchanged.

```
In [29]: df

Out[29]:

   a  b   c   d

0  1  5   5  10

1  1  6   7  14

2  1  7   9  18

3  1  8  11  22

4  1  9  13  26
```

```
In [30]: df.eval("e = a - c", inplace=False)
```

```
Out[30]:
   a  b   c   d   e
0  1  5   5  10  -4
1  1  6   7  14  -6
2  1  7   9  18  -8
3  1  8  11  22 -10
4  1  9  13  26 -12
```

```
In [31]: df
```

```
Out[31]:
   a  b   c   d
0  1  5   5  10
1  1  6   7  14
2  1  7   9  18
3  1  8  11  22
4  1  9  13  26
```

As a convenience, multiple assignments can be performed by using a multi-line string.

```
In [32]: df.eval(
   ....:     """
   ....: c = a + b
   ....: d = a + b + c
   ....: a = 1""",
   ....:     inplace=False,
   ....: )
   ....:

Out[32]:
   a  b   c   d
0  1  5   6  12
1  1  6   7  14
2  1  7   8  16
3  1  8   9  18
4  1  9  10  20
```

The equivalent in standard Python would be

```
In [33]: df = pd.DataFrame(dict(a=range(5), b=range(5, 10)))


In [34]: df["c"] = df["a"] + df["b"]


In [35]: df["d"] = df["a"] + df["b"] + df["c"]
In [36]: df["a"] = 1
```

```
In [37]: df
Out[37]:
   a  b   c   d
0  1  5   5  10
1  1  6   7  14
2  1  7   9  18
3  1  8  11  22
4  1  9  13  26
```

The query method has a inplace keyword which determines whether the query modifies the original frame.

```
In [38]: df = pd.DataFrame(dict(a=range(5), b=range(5, 10)))
```

```
In [39]: df.query("a > 2")
Out[39]:
   a  b
3  3  8
4  4  9
```

```
In [40]: df.query("a > 2", inplace=True)
In [41]: df

Out[41]:

   a  b

3  3  8

4  4  9
```

Local variables

You must *explicitly reference* any local variable that you want to use in an expression by placing the @ character in front of the name. For example,

```
In [42]: df = pd.DataFrame(np.random.randn(5, 2), columns=list("ab"))

In [43]: newcol = np.random.randn(len(df))

In [44]: df.eval("b + @newcol")

Out[44]:

0   -0.173926

1    2.493083

2   -0.881831

3   -0.691045

4    1.334703

dtype: float64


In [45]: df.query("b < @newcol")
```

```
Out[45]:
          a         b
0  0.863987 -0.115998
2 -2.621419 -1.297879
```

If you don’t prefix the local variable with @, pandas will raise an exception telling you the variable is undefined.

When using DataFrame.eval() and DataFrame.query(), this allows you to have a local variable and a DataFrame column with the same name in an expression.

```
In [46]: a = np.random.randn()


In [47]: df.query("@a < a")
Out[47]:
          a         b
0  0.863987 -0.115998


In [48]: df.loc[a < df["a"]]  # same as the previous expression
Out[48]:
          a         b
0  0.863987 -0.115998
```

With pandas.eval() you cannot use the @ prefix *at all*, because it isn’t defined in that context. pandas will let you know this if you try to use @ in a top-level call to pandas.eval(). For example,

```
In [49]: a, b = 1, 2


In [50]: pd.eval("@a + b")

Traceback (most recent call last):


  File /opt/conda/envs/pandas/lib/python3.8/site-packages/IPython/core/interactiveshell.py:3251 in run_code

    exec(code_obj, self.user_global_ns, self.user_ns)


  Input In [50] in <module>

    pd.eval("@a + b")


  File /pandas/pandas/core/computation/eval.py:339 in eval

    _check_for_locals(expr, level, parser)


  File /pandas/pandas/core/computation/eval.py:163 in _check_for_locals

    raise SyntaxError(msg)


  File <string>

SyntaxError: The '@' prefix is not allowed in top-level eval calls.
```

please refer to your variables by name without the '@' prefix.

In this case, you should simply refer to the variables like you would in standard Python.

```
In [51]: pd.eval("a + b")
```

```
Out[51]: 3
```

pandas.eval() parsers

There are two different parsers and two different engines you can use as the backend.

The default 'pandas' parser allows a more intuitive syntax for expressing query-like operations (comparisons, conjunctions and disjunctions). In particular, the precedence of the & and | operators is made equal to the precedence of the corresponding boolean operations and and or.

For example, the above conjunction can be written without parentheses. Alternatively, you can use the 'python' parser to enforce strict Python semantics.

```
In [52]: expr = "(df1 > 0) & (df2 > 0) & (df3 > 0) & (df4 > 0)"
```

```
In [53]: x = pd.eval(expr, parser="python")
```

```
In [54]: expr_no_parens = "df1 > 0 & df2 > 0 & df3 > 0 & df4 > 0"
```

```
In [55]: y = pd.eval(expr_no_parens, parser="pandas")
In [56]: np.all(x == y)
Out[56]: True
```

The same expression can be “anded” together with the word and as well:

```
In [57]: expr = "(df1 > 0) & (df2 > 0) & (df3 > 0) & (df4 > 0)"


In [58]: x = pd.eval(expr, parser="python")

In [59]: expr_with_ands = "df1 > 0 and df2 > 0 and df3 > 0 and df4 > 0"

In [60]: y = pd.eval(expr_with_ands, parser="pandas")

In [61]: np.all(x == y)
Out[61]: True
```

The and and or operators here have the same precedence that they would in vanilla Python.

pandas.eval() backends

There’s also the option to make eval() operate identical to plain ol’ Python.

Note

Using the 'python' engine is generally *not* useful, except for testing other evaluation engines against it. You will achieve no performance benefits using eval() with engine='python' and in fact may incur a performance hit.

You can see this by using pandas.eval() with the 'python' engine. It is a bit slower (not by much) than evaluating the same expression in Python

```
In [62]: %timeit df1 + df2 + df3 + df4
```

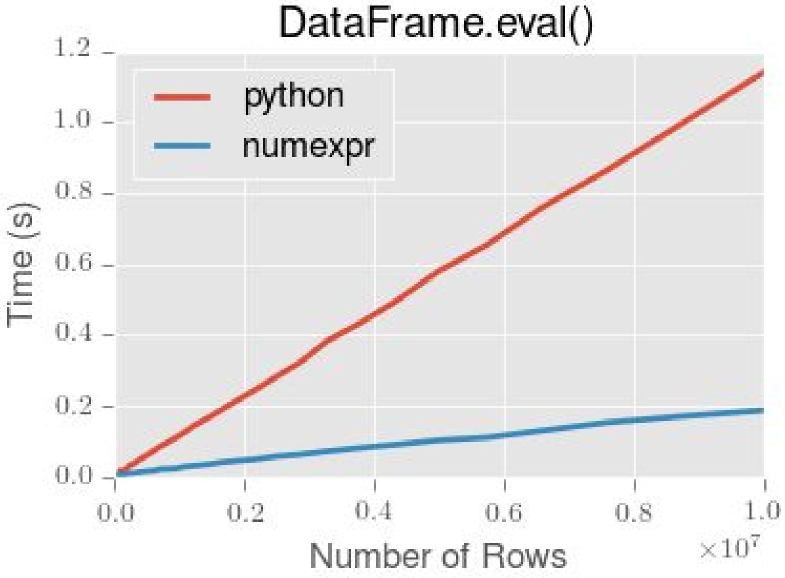
```
8.31 ms +- 544 us per loop (mean +- std. dev. of 7 runs, 100 loops each)
```

```
In [63]: %timeit pd.eval("df1 + df2 + df3 + df4", engine="python")
```

```
9.31 ms +- 354 us per loop (mean +- std. dev. of 7 runs, 100 loops each)
```

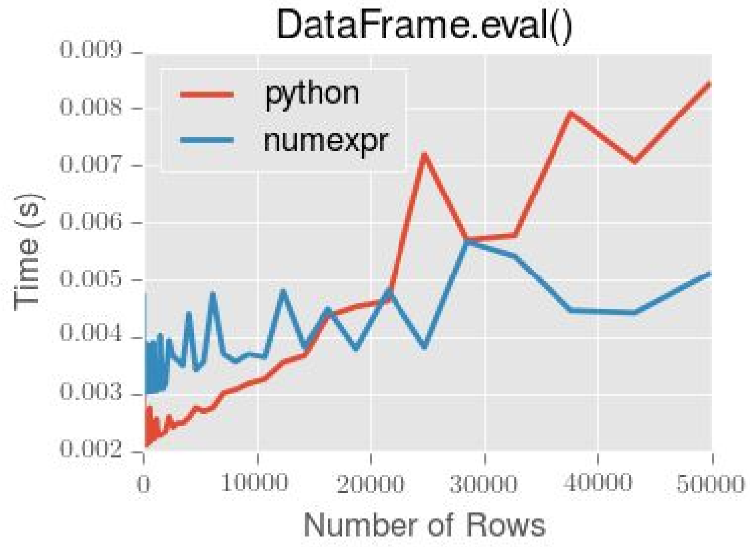
pandas.eval() performance

eval() is intended to speed up certain kinds of operations. In particular, those operations involving complex expressions with large DataFrame/Series objects should see a significant performance benefit. Here is a plot showing the running time of pandas.eval() as function of the size of the frame involved in the computation. The two lines are two different engines.



Note

Operations with smallish objects (around 15k-20k rows) are faster using plain Python:



This plot was created using a DataFrame with 3 columns each containing floating point values generated using numpy.random.randn().

Technical minutia regarding expression evaluation

Expressions that would result in an object dtype or involve datetime operations (because of NaT) must be evaluated in Python space. The main reason for this behavior is to maintain backwards compatibility with versions of NumPy < 1.7. In those versions of NumPy a call to ndarray.astype(str) will truncate any strings that are more than 60 characters in length. Second, we can't pass object arrays to numexpr thus string comparisons must be evaluated in Python space.

The upshot is that this *only* applies to object-dtype expressions. So, if you have an expression–for example

```
In [64]: df = pd.DataFrame(
   ....:     {"strings": np.repeat(list("cba"), 3), "nums": np.repeat(range(3), 3)}
   ....: )
   ....:
```

```
In [65]: df

Out[65]:

  strings  nums
0       c     0
1       c     0
2       c     0
3       b     1
4       b     1
5       b     1
6       a     2
7       a     2
8       a     2


In [66]: df.query("strings == 'a' and nums == 1")

Out[66]:

Empty DataFrame

Columns: [strings, nums]

Index: []
```

the numeric part of the comparison (nums == 1) will be evaluated by numexpr.

In general, DataFrame.query()/pandas.eval() will evaluate the subexpressions that *can* be evaluated by numexpr and those that must be evaluated in

Python space transparently to the user. This is done by inferring the result type of an expression from its arguments and operators.

CHEATSHEET

Python For Data Science Cheat Sheet: Pandas Basics

Use the following import convention:

```
import pandas as pd
```

Pandas Data Structures

Series
A one-dimensional labeled array capable of holding any data type

```
s = pd.Series([3, -5, 7, 4], index=['a', 'b', 'c', 'd'])
```

A	3

DataFrame
A two-dimensional labeled data structure with columns of potentially different types

```
data = {'Country': ['Belgium', 'India', 'Brazil'],
```

```
'Capital': ['Brussels', 'New Delhi', 'Brasilia'],
'Population': [11190846, 1303171035, 207847528]}
```

```
df = pd.DataFrame(data,columns=['Country', 'Capital', 'Population'])
```

```
     Country   Capital     Population
1    Belgium   Brussels    11190846
2    India     New Delhi   1303171035
3    Brazil    Brasilia    207847528
```

Please note that the first column 1,2,3 is the index and Country,Capital,Population are the Columns.

Asking For Help

```
help(pd.Series.loc)
```

I/O

Read and Write to CSV

```
pd.read_csv('file.csv', header=None, nrows=5)
```

```
df.to_csv('myDataFrame.csv')
```

Read multiple sheets from the same file

```
xlsx = pd.ExcelFile('file.xls')
```

```
df = pd.read_excel(xlsx, 'Sheet1')
```

Read and Write to Excel

```
pd.read_excel('file.xlsx')
```

```
df.to_excel('dir/myDataFrame.xlsx',  sheet_name='Sheet1')
```

Read and Write to SQL Query or Database Table
(read_sql()is a convenience wrapper around read_sql_table() and read-_sql_query())

```
from sqlalchemy import create_engine
```

```
engine = create_engine('sqlite:///:memory:')
```

```
pd.read_sql(SELECT * FROM my_table;, engine)
```

```
pd.read_sql_table('my_table', engine)
```

```
pd.read_sql_query(SELECT * FROM my_table;', engine)
```

```
df.to_sql('myDf', engine)
```

Selection
Getting
Get one element

```
s['b']
```

```
-5
```

Get subset of a DataFrame

```
df[1:]
```

```
Country     Capital   Population
1  India    New Delhi 1303171035
2  Brazil   Brasilia  207847528
```

Selecting', Boolean Indexing and Setting
By Position
Select single value by row and and column

```
df.iloc([0], [0])
```

```
'Belgium'
```

```
df.iat([0], [0])
```

```
'Belgium'
```

By Label
Select single value by row and column labels

```
df.loc([0],  ['Country'])
```

```
'Belgium'
```

```
df.at([0],  ['Country'])
```

```
'Belgium'
```

By Label/Position
Select single row of subset of rows

```
df.ix[2]
```

```
Country      Brazil
Capital    Brasilia
Population  207847528
```

Select a single column of subset of columns

```
df.ix[:, 'Capital']
```

```
0     Brussels
1    New Delhi
2     Brasilia
```

Select rows and columns

```
df.ix[1, 'Capital']
```

```
'New Delhi'
```

Boolean Indexing
Series s where value is not >1

```
s[~(s > 1)]
```

s where value is <-1 or >2

```
s[(s < -1) | (s > 2)]
```

Use filter to adjust DataFrame

```
df[df['Population']>1200000000]
```

Setting
Set index a of Series s to 6

```
s['a'] = 6
```

Dropping
Drop values from rows (axis=0)

```
s.drop(['a',  'c'])
```

Drop values from columns(axis=1)

```
df.drop('Country', axis=1)
```

Sort and Rank
Sort by labels along an axis

```
df.sort_index()
```

Sort by the values along an axis

```
df.sort_values(by='Country')
```

Assign ranks to entries

```
df.rank()
```

Retrieving Series/DataFrame Information
Basic Information

(rows, columns)

```
df.shape
```

Describe index

```
df.index
```

Describe DataFrame columns

```
df.columns
```

Info on DataFrame

```
df.info()
```

Number of non-NA values

```
df.count()
```

Summary
Sum of values

```
df.sum()
```

Cumulative sum of values

```
df.cumsum()
```

Minimum/maximum values

```
df.min()/df.max()
```

Minimum/Maximum index value

```
df.idxmin()/df.idxmax()
```

Summary statistics

```
df.describe()
```

Mean of values

```
df.mean()
```

Median of values

```
df.median()
```

Applying Functions

```
f = lambda x: x*2
```

Apply function

```
df.apply(f)
```

Apply function element-wise

```
df.applynap(f)
```

Internal Data Alignment
NA values are introduced in the indices that don't overlap:

```
s3 = pd.Series([7, -2, 3],  index=['a',  'c',  'd'])
```

```
s + s3
```

```
a    10.0

b    NaN

c    5.0

d    7.0
```

Arithmetic Operations with Fill Methods
You can also do the internal data alignment yourself with the help of the fill methods:

```
s.add(s3, fill_value=0)
```

```
a   10.0

b   -5.0

c   5.0

d   7.0
```

```
s.sub(s3, fill_value=2)
```

```
s.div(s3, fill_value=4)
```

```
s.mul(s3, fill_value=3)
```