Depending on df construction, `ignore_index` may be needed

```
In [180]: df = df1.append(df2, ignore_index=True)

In [181]: df
Out[181]:
           A         B         C
0  -0.870117 -0.479265 -0.790855
1   0.144817  1.726395 -0.464535
2  -0.821906  1.597605  0.187307
3  -0.128342 -1.511638 -0.289858
4   0.399194 -1.430030 -0.639760
5   1.115116 -2.012600  1.810662
6  -0.870117 -0.479265 -0.790855
7   0.144817  1.726395 -0.464535
8  -0.821906  1.597605  0.187307
9  -0.128342 -1.511638 -0.289858
10  0.399194 -1.430030 -0.639760
11  1.115116 -2.012600  1.810662
```

Self Join of a DataFrame

```
In [182]: df = pd.DataFrame(data={'Area': ['A'] * 5 + ['C'] * 2,
   .....:                         'Bins': [110] * 2 + [160] * 3 + [40] * 2,
   .....:                         'Test_0': [0, 1, 0, 1, 2, 0, 1],
   .....:                         'Data': np.random.randn(7)})
   .....:

In [183]: df
Out[183]:
  Area  Bins  Test_0      Data
0    A   110       0 -0.433937
1    A   110       1 -0.160552
2    A   160       0  0.744434
3    A   160       1  1.754213
4    A   160       2  0.000850
5    C    40       0  0.342243
6    C    40       1  1.070599

In [184]: df['Test_1'] = df['Test_0'] - 1

In [185]: pd.merge(df, df, left_on=['Bins', 'Area', 'Test_0'],
   .....:          right_on=['Bins', 'Area', 'Test_1'],
   .....:          suffixes=('_L', '_R'))
   .....:
Out[185]:
  Area  Bins  Test_0_L    Data_L  Test_1_L  Test_0_R    Data_R  Test_1_R
0    A   110         0 -0.433937        -1         1 -0.160552         0
1    A   160         0  0.744434        -1         1  1.754213         0
2    A   160         1  1.754213         0         2  0.000850         1
3    C    40         0  0.342243        -1         1  1.070599         0
```

How to set the index and join

KDB like asof join

Join with a criteria based on the values

Using searchsorted to merge based on values inside a range

2.22.8 Plotting

The *Plotting* docs.

Make Matplotlib look like R

Setting x-axis major and minor labels

Plotting multiple charts in an ipython notebook

Creating a multi-line plot

Plotting a heatmap

Annotate a time-series plot

Annotate a time-series plot #2

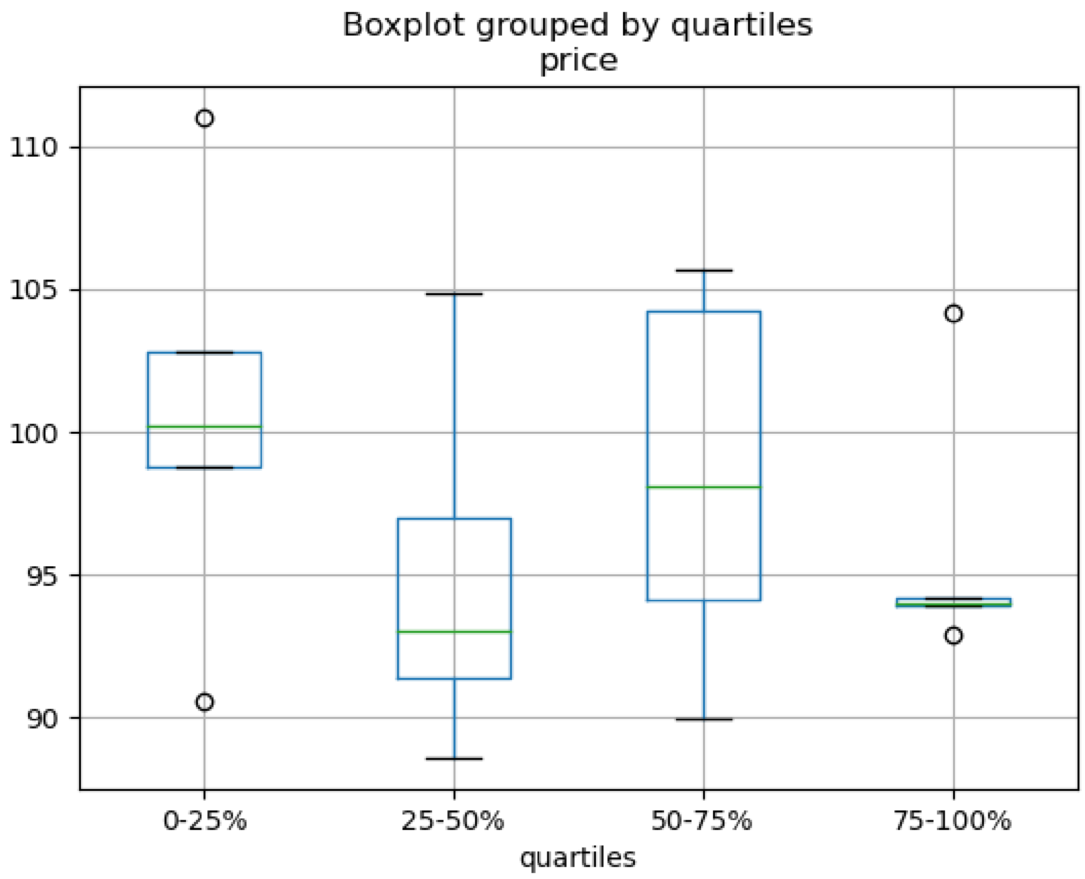
Generate Embedded plots in excel files using Pandas, Vincent and xlsxwriter

Boxplot for each quartile of a stratifying variable

```
In [186]: df = pd.DataFrame(
   .....:     {'stratifying_var': np.random.uniform(0, 100, 20),
   .....:      'price': np.random.normal(100, 5, 20)})
   .....:

In [187]: df['quartiles'] = pd.qcut(
   .....:     df['stratifying_var'],
   .....:     4,
   .....:     labels=['0-25%', '25-50%', '50-75%', '75-100%'])
   .....:

In [188]: df.boxplot(column='price', by='quartiles')
Out[188]: <matplotlib.axes._subplots.AxesSubplot at 0x7f533d322b90>
```



2.22.9 Data In/Out

Performance comparison of SQL vs HDF5

CSV

The *CSV* docs

read_csv in action

appending to a csv

Reading a csv chunk-by-chunk

Reading only certain rows of a csv chunk-by-chunk

Reading the first few lines of a frame

Reading a file that is compressed but not by `gzip/bz2` (the native compressed formats which `read_csv` understands). This example shows a `WinZipped` file, but is a general application of opening the file within a context manager and using that handle to read. See here

Inferring dtypes from a file

Dealing with bad lines

Dealing with bad lines II

Reading CSV with Unix timestamps and converting to local timezone

Write a multi-row index CSV without writing duplicates

Reading multiple files to create a single DataFrame

The best way to combine multiple files into a single DataFrame is to read the individual frames one by one, put all of the individual frames into a list, and then combine the frames in the list using `pd.concat()`:

```
In [189]: for i in range(3):
   .....:     data = pd.DataFrame(np.random.randn(10, 4))
   .....:     data.to_csv('file_{}.csv'.format(i))
   .....:

In [190]: files = ['file_0.csv', 'file_1.csv', 'file_2.csv']

In [191]: result = pd.concat([pd.read_csv(f) for f in files], ignore_index=True)
```

You can use the same approach to read all files matching a pattern. Here is an example using `glob`:

```
In [192]: import glob

In [193]: import os

In [194]: files = glob.glob('file_*.csv')

In [195]: result = pd.concat([pd.read_csv(f) for f in files], ignore_index=True)
```

Finally, this strategy will work with the other `pd.read_*(...)` functions described in the *io docs*.

Parsing date components in multi-columns

Parsing date components in multi-columns is faster with a format

```
In [196]: i = pd.date_range('20000101', periods=10000)

In [197]: df = pd.DataFrame({'year': i.year, 'month': i.month, 'day': i.day})

In [198]: df.head()
Out[198]:
   year  month  day
0  2000      1    1
1  2000      1    2
2  2000      1    3
3  2000      1    4
4  2000      1    5

In [199]: %timeit pd.to_datetime(df.year * 10000 + df.month * 100 + df.day, format='%Y
↪%m%d')
   .....: ds = df.apply(lambda x: "%04d%02d%02d" % (x['year'],
   .....:                                           x['month'], x['day']), axis=1)
   .....: ds.head()
   .....: %timeit pd.to_datetime(ds)
   .....:
```

(continues on next page)

(continued from previous page)

```
15.6 ms +- 1.54 ms per loop (mean +- std. dev. of 7 runs, 10 loops each)
4.1 ms +- 497 us per loop (mean +- std. dev. of 7 runs, 100 loops each)
```

Skip row between header and data

```
In [200]: data = """;;;;
   .....:  ;;;;
   .....:  ;;;;
   .....:  ;;;;
   .....:  ;;;;
   .....:  ;;;;
   .....: ;;;;
   .....:  ;;;;
   .....:  ;;;;
   .....: ;;;;
   .....: date;Param1;Param2;Param4;Param5
   .....:     ;m²;°C;m²;m
   .....: ;;;;
   .....: 01.01.1990 00:00;1;1;2;3
   .....: 01.01.1990 01:00;5;3;4;5
   .....: 01.01.1990 02:00;9;5;6;7
   .....: 01.01.1990 03:00;13;7;8;9
   .....: 01.01.1990 04:00;17;9;10;11
   .....: 01.01.1990 05:00;21;11;12;13
   .....: """
   .....:
```

Option 1: pass rows explicitly to skip rows

```
In [201]: from io import StringIO

In [202]: pd.read_csv(StringIO(data), sep=';', skiprows=[11, 12],
   .....:             index_col=0, parse_dates=True, header=10)
   .....:
Out[202]:
                     Param1  Param2  Param4  Param5
date
1990-01-01 00:00:00       1       1       2       3
1990-01-01 01:00:00       5       3       4       5
1990-01-01 02:00:00       9       5       6       7
1990-01-01 03:00:00      13       7       8       9
1990-01-01 04:00:00      17       9      10      11
1990-01-01 05:00:00      21      11      12      13
```

Option 2: read column names and then data

```
In [203]: pd.read_csv(StringIO(data), sep=';', header=10, nrows=10).columns
Out[203]: Index(['date', 'Param1', 'Param2', 'Param4', 'Param5'], dtype='object')

In [204]: columns = pd.read_csv(StringIO(data), sep=';', header=10, nrows=10).columns

In [205]: pd.read_csv(StringIO(data), sep=';', index_col=0,
   .....:             header=12, parse_dates=True, names=columns)
   .....:
Out[205]:
                     Param1  Param2  Param4  Param5
date
1990-01-01 00:00:00       1       1       2       3
1990-01-01 01:00:00       5       3       4       5
1990-01-01 02:00:00       9       5       6       7
1990-01-01 03:00:00      13       7       8       9
1990-01-01 04:00:00      17       9      10      11
1990-01-01 05:00:00      21      11      12      13
```

SQL

The *SQL* docs

Reading from databases with SQL

Excel

The *Excel* docs

Reading from a filelike handle

Modifying formatting in XlsxWriter output

HTML

Reading HTML tables from a server that cannot handle the default request header

HDFStore

The *HDFStores* docs

Simple queries with a Timestamp Index

Managing heterogeneous data using a linked multiple table hierarchy

Merging on-disk tables with millions of rows

Avoiding inconsistencies when writing to a store from multiple processes/threads

De-duplicating a large store by chunks, essentially a recursive reduction operation. Shows a function for taking in data from csv file and creating a store by chunks, with date parsing as well. See here

Creating a store chunk-by-chunk from a csv file

Appending to a store, while creating a unique index

Large Data work flows

Reading in a sequence of files, then providing a global unique index to a store while appending

Groupby on a HDFStore with low group density

Groupby on a HDFStore with high group density

Hierarchical queries on a HDFStore

Counting with a HDFStore

Troubleshoot HDFStore exceptions

Setting min_itemsize with strings

Using ptrepack to create a completely-sorted-index on a store

Storing Attributes to a group node

```
In [206]: df = pd.DataFrame(np.random.randn(8, 3))

In [207]: store = pd.HDFStore('test.h5')

In [208]: store.put('df', df)

# you can store an arbitrary Python object via pickle
In [209]: store.get_storer('df').attrs.my_attribute = {'A': 10}

In [210]: store.get_storer('df').attrs.my_attribute
Out[210]: {'A': 10}
```

Binary files

pandas readily accepts NumPy record arrays, if you need to read in a binary file consisting of an array of C structs. For example, given this C program in a file called `main.c` compiled with `gcc main.c -std=gnu99` on a 64-bit machine,

```
#include <stdio.h>
#include <stdint.h>

typedef struct _Data
{
    int32_t count;
    double avg;
    float scale;
} Data;

int main(int argc, const char *argv[])
{
    size_t n = 10;
    Data d[n];

    for (int i = 0; i < n; ++i)
    {
        d[i].count = i;
        d[i].avg = i + 1.0;
        d[i].scale = (float) i + 2.0f;
    }
```

(continues on next page)

(continued from previous page)

```
    FILE *file = fopen("binary.dat", "wb");
    fwrite(&d, sizeof(Data), n, file);
    fclose(file);

    return 0;
}
```

the following Python code will read the binary file `'binary.dat'` into a pandas `DataFrame`, where each element of the struct corresponds to a column in the frame:

```
names = 'count', 'avg', 'scale'

# note that the offsets are larger than the size of the type because of
# struct padding
offsets = 0, 8, 16
formats = 'i4', 'f8', 'f4'
dt = np.dtype({'names': names, 'offsets': offsets, 'formats': formats},
              align=True)
df = pd.DataFrame(np.fromfile('binary.dat', dt))
```

Note: The offsets of the structure elements may be different depending on the architecture of the machine on which the file was created. Using a raw binary file format like this for general data storage is not recommended, as it is not cross platform. We recommended either HDF5 or parquet, both of which are supported by pandas' IO facilities.

2.22.10 Computation

Numerical integration (sample-based) of a time series

Correlation

Often it's useful to obtain the lower (or upper) triangular form of a correlation matrix calculated from *DataFrame.corr()*. This can be achieved by passing a boolean mask to `where` as follows:

```
In [211]: df = pd.DataFrame(np.random.random(size=(100, 5)))

In [212]: corr_mat = df.corr()

In [213]: mask = np.tril(np.ones_like(corr_mat, dtype=np.bool), k=-1)

In [214]: corr_mat.where(mask)
Out[214]:
          0         1         2         3   4
0       NaN       NaN       NaN       NaN NaN
1 -0.018923       NaN       NaN       NaN NaN
2 -0.076296 -0.012464       NaN       NaN NaN
3 -0.169941 -0.289416  0.076462       NaN NaN
4  0.064326  0.018759 -0.084140 -0.079859 NaN
```

The *method* argument within *DataFrame.corr* can accept a callable in addition to the named correlation types. Here we compute the distance correlation matrix for a *DataFrame* object.

```
In [215]: def distcorr(x, y):
   .....:     n = len(x)
   .....:     a = np.zeros(shape=(n, n))
   .....:     b = np.zeros(shape=(n, n))
   .....:     for i in range(n):
   .....:         for j in range(i + 1, n):
   .....:             a[i, j] = abs(x[i] - x[j])
   .....:             b[i, j] = abs(y[i] - y[j])
   .....:     a += a.T
   .....:     b += b.T
   .....:     a_bar = np.vstack([np.nanmean(a, axis=0)] * n)
   .....:     b_bar = np.vstack([np.nanmean(b, axis=0)] * n)
   .....:     A = a - a_bar - a_bar.T + np.full(shape=(n, n), fill_value=a_bar.mean())
   .....:     B = b - b_bar - b_bar.T + np.full(shape=(n, n), fill_value=b_bar.mean())
   .....:     cov_ab = np.sqrt(np.nansum(A * B)) / n
   .....:     std_a = np.sqrt(np.sqrt(np.nansum(A**2)) / n)
   .....:     std_b = np.sqrt(np.sqrt(np.nansum(B**2)) / n)
   .....:     return cov_ab / std_a / std_b
   .....:

In [216]: df = pd.DataFrame(np.random.normal(size=(100, 3)))

In [217]: df.corr(method=distcorr)
Out[217]:
          0         1         2
0  1.000000  0.199653  0.214871
1  0.199653  1.000000  0.195116
2  0.214871  0.195116  1.000000
```

2.22.11 Timedeltas

The *Timedeltas* docs.

Using timedeltas

```
In [218]: import datetime

In [219]: s = pd.Series(pd.date_range('2012-1-1', periods=3, freq='D'))

In [220]: s - s.max()
Out[220]:
0   -2 days
1   -1 days
2    0 days
dtype: timedelta64[ns]

In [221]: s.max() - s
Out[221]:
0   2 days
1   1 days
2   0 days
dtype: timedelta64[ns]

In [222]: s - datetime.datetime(2011, 1, 1, 3, 5)
Out[222]:
0   364 days 20:55:00
```

(continues on next page)

(continued from previous page)

```
1   365 days 20:55:00
2   366 days 20:55:00
dtype: timedelta64[ns]

In [223]: s + datetime.timedelta(minutes=5)
Out[223]:
0   2012-01-01 00:05:00
1   2012-01-02 00:05:00
2   2012-01-03 00:05:00
dtype: datetime64[ns]

In [224]: datetime.datetime(2011, 1, 1, 3, 5) - s
Out[224]:
0   -365 days +03:05:00
1   -366 days +03:05:00
2   -367 days +03:05:00
dtype: timedelta64[ns]

In [225]: datetime.timedelta(minutes=5) + s
Out[225]:
0   2012-01-01 00:05:00
1   2012-01-02 00:05:00
2   2012-01-03 00:05:00
dtype: datetime64[ns]
```

Adding and subtracting deltas and dates

```
In [226]: deltas = pd.Series([datetime.timedelta(days=i) for i in range(3)])

In [227]: df = pd.DataFrame({'A': s, 'B': deltas})

In [228]: df
Out[228]:
           A      B
0 2012-01-01 0 days
1 2012-01-02 1 days
2 2012-01-03 2 days

In [229]: df['New Dates'] = df['A'] + df['B']

In [230]: df['Delta'] = df['A'] - df['New Dates']

In [231]: df
Out[231]:
           A      B  New Dates   Delta
0 2012-01-01 0 days 2012-01-01  0 days
1 2012-01-02 1 days 2012-01-03 -1 days
2 2012-01-03 2 days 2012-01-05 -2 days

In [232]: df.dtypes
Out[232]:
A              datetime64[ns]
B             timedelta64[ns]
New Dates      datetime64[ns]
Delta         timedelta64[ns]
dtype: object
```

Another example

Values can be set to NaT using np.nan, similar to datetime

```
In [233]: y = s - s.shift()

In [234]: y
Out[234]:
0      NaT
1   1 days
2   1 days
dtype: timedelta64[ns]

In [235]: y[1] = np.nan

In [236]: y
Out[236]:
0      NaT
1      NaT
2   1 days
dtype: timedelta64[ns]
```

2.22.12 Aliasing axis names

To globally provide aliases for axis names, one can define these 2 functions:

```
In [237]: def set_axis_alias(cls, axis, alias):
   .....:     if axis not in cls._AXIS_NUMBERS:
   .....:         raise Exception("invalid axis [%s] for alias [%s]" % (axis, alias))
   .....:     cls._AXIS_ALIASES[alias] = axis
   .....:
```

```
In [238]: def clear_axis_alias(cls, axis, alias):
   .....:     if axis not in cls._AXIS_NUMBERS:
   .....:         raise Exception("invalid axis [%s] for alias [%s]" % (axis, alias))
   .....:     cls._AXIS_ALIASES.pop(alias, None)
   .....:
```

```
In [239]: set_axis_alias(pd.DataFrame, 'columns', 'myaxis2')

In [240]: df2 = pd.DataFrame(np.random.randn(3, 2), columns=['c1', 'c2'],
   .....:                    index=['i1', 'i2', 'i3'])
   .....:

In [241]: df2.sum(axis='myaxis2')
Out[241]:
i1   -0.461013
i2    2.040016
i3    0.904681
dtype: float64

In [242]: clear_axis_alias(pd.DataFrame, 'columns', 'myaxis2')
```

2.22.13 Creating example data

To create a dataframe from every combination of some given values, like R's `expand.grid()` function, we can create a dict where the keys are column names and the values are lists of the data values:

```
In [243]: def expand_grid(data_dict):
   .....:     rows = itertools.product(*data_dict.values())
   .....:     return pd.DataFrame.from_records(rows, columns=data_dict.keys())
   .....:

In [244]: df = expand_grid({'height': [60, 70],
   .....:                   'weight': [100, 140, 180],
   .....:                   'sex': ['Male', 'Female']})
   .....:

In [245]: df
Out[245]:
    height  weight     sex
0       60     100    Male
1       60     100  Female
2       60     140    Male
3       60     140  Female
4       60     180    Male
5       60     180  Female
6       70     100    Male
7       70     100  Female
8       70     140    Male
9       70     140  Female
10      70     180    Male
11      70     180  Female
```

CHAPTER

THREE

API REFERENCE

This page gives an overview of all public pandas objects, functions and methods. All classes and functions exposed in `pandas.*` namespace are public.

Some subpackages are public which include `pandas.errors`, `pandas.plotting`, and `pandas.testing`. Public functions in `pandas.io` and `pandas.tseries` submodules are mentioned in the documentation. `pandas.api.types` subpackage holds some public functions related to data types in pandas.

> **Warning:** The `pandas.core`, `pandas.compat`, and `pandas.util` top-level modules are PRIVATE. Stable functionality in such modules is not guaranteed.

3.1 Input/output

3.1.1 Pickling

read_pickle(filepath_or_buffer, . . .)	Load pickled pandas object (or any object) from file.

pandas.read_pickle

`pandas.`**`read_pickle`**(*filepath_or_buffer: Union[str, pathlib.Path, IO[~ AnyStr]], compression: Union[str, NoneType] = 'infer'*)

Load pickled pandas object (or any object) from file.

> **Warning:** Loading pickled data received from untrusted sources can be unsafe. See here.

Parameters

filepath_or_buffer [str, path object or file-like object] File path, URL, or buffer where the pickled object will be loaded from.

Changed in version 1.0.0: Accept URL. URL is not limited to S3 and GCS.

compression [{'infer', 'gzip', 'bz2', 'zip', 'xz', None}, default 'infer'] If 'infer' and 'path_or_url' is path-like, then detect compression from the following extensions: '.gz', '.bz2', '.zip', or '.xz' (otherwise no compression) If 'infer' and 'path_or_url' is not path-like, then use None (= no decompression).

Returns

unpickled [same type as object stored in file]

See also:

DataFrame.to_pickle Pickle (serialize) DataFrame object to file.

Series.to_pickle Pickle (serialize) Series object to file.

read_hdf Read HDF5 file into a DataFrame.

read_sql Read SQL query or database table into a DataFrame.

read_parquet Load a parquet object, returning a DataFrame.

Notes

read_pickle is only guaranteed to be backwards compatible to pandas 0.20.3.

Examples

```
>>> original_df = pd.DataFrame({"foo": range(5), "bar": range(5, 10)})
>>> original_df
   foo  bar
0    0    5
1    1    6
2    2    7
3    3    8
4    4    9
>>> pd.to_pickle(original_df, "./dummy.pkl")
```

```
>>> unpickled_df = pd.read_pickle("./dummy.pkl")
>>> unpickled_df
   foo  bar
0    0    5
1    1    6
2    2    7
3    3    8
4    4    9
```

```
>>> import os
>>> os.remove("./dummy.pkl")
```

3.1.2 Flat file

read_table(filepath_or_buffer, pathlib.Path, . . .)	Read general delimited file into DataFrame.
read_csv(filepath_or_buffer, pathlib.Path, . . .)	Read a comma-separated values (csv) file into DataFrame.
read_fwf(filepath_or_buffer, pathlib.Path, . . .)	Read a table of fixed-width formatted lines into DataFrame.

pandas.read_table

`pandas.`**`read_table`**(*filepath_or_buffer: Union[str, pathlib.Path, IO[~ AnyStr]], sep='\t', delimiter=None, header='infer', names=None, index_col=None, usecols=None, squeeze=False, prefix=None, mangle_dupe_cols=True, dtype=None, engine=None, converters=None, true_values=None, false_values=None, skipinitialspace=False, skiprows=None, skipfooter=0, nrows=None, na_values=None, keep_default_na=True, na_filter=True, verbose=False, skip_blank_lines=True, parse_dates=False, infer_datetime_format=False, keep_date_col=False, date_parser=None, dayfirst=False, cache_dates=True, iterator=False, chunksize=None, compression='infer', thousands=None, decimal: str = '.', lineterminator=None, quotechar='"', quoting=0, doublequote=True, escapechar=None, comment=None, encoding=None, dialect=None, error_bad_lines=True, warn_bad_lines=True, delim_whitespace=False, low_memory=True, memory_map=False, float_precision=None*)

Read general delimited file into DataFrame.

Also supports optionally iterating or breaking of the file into chunks.

Additional help can be found in the online docs for IO Tools.

Parameters

filepath_or_buffer [str, path object or file-like object] Any valid string path is acceptable. The string could be a URL. Valid URL schemes include http, ftp, s3, and file. For file URLs, a host is expected. A local file could be: file://localhost/path/to/table.csv.

If you want to pass in a path object, pandas accepts any `os.PathLike`.

By file-like object, we refer to objects with a `read()` method, such as a file handler (e.g. via builtin `open` function) or `StringIO`.

sep [str, default '\t' (tab-stop)] 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'`.

delimiter [str, default `None`] Alias for sep.

header [int, list of int, 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 integers that specify row locations for a multi-index 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.

names [array-like, optional] List of column names to use. If the file contains a header row, then you should explicitly pass `header=0` to override the column names. Duplicates in this list are not allowed.

index_col [int, str, sequence of int / str, or False, 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.

usecols [list-like or callable, optional] 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). 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. An example of a valid callable argument would be `lambda x: x.upper() in ['AAA', 'BBB', 'DDD']`. Using this parameter results in much faster parsing time and lower memory usage.

squeeze [bool, default False] If the parsed data only contains one column then return a Series.

prefix [str, optional] Prefix to add to column numbers when no header, e.g. 'X' for X0, X1, …

mangle_dupe_cols [bool, 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.

dtype [Type name or dict of column -> type, optional] Data type for data or columns. E.g. {'a': np.float64, 'b': np.int32, 'c': 'Int64'} Use *str* or *object* together with suitable *na_values* settings to preserve and not interpret dtype. If converters are specified, they will be applied INSTEAD of dtype conversion.

engine [{'c', 'python'}, optional] Parser engine to use. The C engine is faster while the python engine is currently more feature-complete.

converters [dict, optional] Dict of functions for converting values in certain columns. Keys can either be integers or column labels.

true_values [list, optional] Values to consider as True.

false_values [list, optional] Values to consider as False.

skipinitialspace [bool, default False] Skip spaces after delimiter.

skiprows [list-like, int or callable, optional] 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. An example of a valid callable argument would be `lambda x: x in [0, 2]`.

skipfooter [int, default 0] Number of lines at bottom of file to skip (Unsupported with engine='c').

nrows [int, optional] Number of rows of file to read. Useful for reading pieces of large files.

na_values [scalar, str, list-like, or dict, optional] Additional strings to recognize as NA/NaN. If dict passed, specific per-column NA values. By default the following values are interpreted as NaN: '', '#N/A', '#N/A N/A', '#NA', '-1.#IND', '-1.#QNAN', '-NaN', '-nan', '1.#IND', '1.#QNAN', '<NA>', 'N/A', 'NA', 'NULL', 'NaN', 'n/a', 'nan', 'null'.

keep_default_na [bool, 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_filter [bool, 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.

verbose [bool, default False] Indicate number of NA values placed in non-numeric columns.

skip_blank_lines [bool, default True] If True, skip over blank lines rather than interpreting as NaN values.

parse_dates [bool or list of int or names or list of lists or dict, default False] The behavior is as follows:

- boolean. If True -> try parsing the index.
- list of int or names. e.g. If [1, 2, 3] -> try parsing columns 1, 2, 3 each as a separate date column.
- list of lists. e.g. If [[1, 3]] -> combine columns 1 and 3 and parse as a single date column.
- dict, e.g. {'foo' : [1, 3]} -> parse columns 1, 3 as date and call result 'foo'

If a column or index cannot be represented as an array of datetimes, say because of an unparseable value or a mixture of timezones, the column or index will be returned unaltered as an object data type. For non-standard datetime parsing, use `pd.to_datetime` after `pd.read_csv`. To parse an index or column with a mixture of timezones, specify `date_parser` to be a partially-applied *`pandas.to_datetime()`* with `utc=True`. See *Parsing a CSV with mixed timezones* for more.

Note: A fast-path exists for iso8601-formatted dates.

infer_datetime_format [bool, default False] If True and *parse_dates* is enabled, pandas will attempt to infer the format of the datetime strings in the columns, and if it can be inferred, switch to a faster method of parsing them. In some cases this can increase the parsing speed by 5-10x.

keep_date_col [bool, default False] If True and *parse_dates* specifies combining multiple columns then keep the original columns.

date_parser [function, optional] 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.

dayfirst [bool, default False] DD/MM format dates, international and European format.

cache_dates [bool, 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.

iterator [bool, default False] Return TextFileReader object for iteration or getting chunks with `get_chunk()`.

chunksize [int, optional] Return TextFileReader object for iteration. See the IO Tools docs for more information on `iterator` and `chunksize`.

compression [{'infer', 'gzip', 'bz2', 'zip', 'xz', None}, default 'infer'] For on-the-fly decompression of on-disk data. If 'infer' and *filepath_or_buffer* is path-like, then detect compression from the following extensions: '.gz', '.bz2', '.zip', or '.xz' (otherwise no decompression). If using 'zip', the ZIP file must contain only one data file to be read in. Set to None for no decompression.

thousands [str, optional] Thousands separator.

decimal [str, default '.'] Character to recognize as decimal point (e.g. use ',' for European data).

lineterminator [str (length 1), optional] Character to break file into lines. Only valid with C parser.

quotechar [str (length 1), optional] The character used to denote the start and end of a quoted item. Quoted items can include the delimiter and it will be ignored.

quoting [int 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).

doublequote [bool, 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.

escapechar [str (length 1), optional] One-character string used to escape other characters.

comment [str, optional] 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.

encoding [str, optional] Encoding to use for UTF when reading/writing (ex. 'utf-8'). List of Python standard encodings .

dialect [str or csv.Dialect, optional] 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_bad_lines [bool, default True] 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.

warn_bad_lines [bool, default True] If error_bad_lines is False, and warn_bad_lines is True, a warning for each "bad line" will be output.

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

low_memory [bool, 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_map [bool, 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.

float_precision [str, optional] 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.

Returns

DataFrame or TextParser A comma-separated values (csv) file is returned as two-dimensional data structure with labeled axes.

See also:

`to_csv` Write DataFrame to a comma-separated values (csv) file.

`read_csv` Read a comma-separated values (csv) file into DataFrame.

`read_fwf` Read a table of fixed-width formatted lines into DataFrame.

Examples

```
>>> pd.read_table('data.csv')
```

pandas.read_csv

`pandas.`**`read_csv`**(*filepath_or_buffer: Union[str, pathlib.Path, IO[~ AnyStr]], sep=',', delimiter=None, header='infer', names=None, index_col=None, usecols=None, squeeze=False, prefix=None, mangle_dupe_cols=True, dtype=None, engine=None, converters=None, true_values=None, false_values=None, skipinitialspace=False, skiprows=None, skipfooter=0, nrows=None, na_values=None, keep_default_na=True, na_filter=True, verbose=False, skip_blank_lines=True, parse_dates=False, infer_datetime_format=False, keep_date_col=False, date_parser=None, dayfirst=False, cache_dates=True, iterator=False, chunksize=None, compression='infer', thousands=None, decimal: str = '.', lineterminator=None, quotechar='"', quoting=0, doublequote=True, escapechar=None, comment=None, encoding=None, dialect=None, error_bad_lines=True, warn_bad_lines=True, delim_whitespace=False, low_memory=True, memory_map=False, float_precision=None*)

Read a comma-separated values (csv) file into DataFrame.

Also supports optionally iterating or breaking of the file into chunks.

Additional help can be found in the online docs for IO Tools.

Parameters

filepath_or_buffer [str, path object or file-like object] Any valid string path is acceptable. The string could be a URL. Valid URL schemes include http, ftp, s3, and file. For file URLs, a host is expected. A local file could be: file://localhost/path/to/table.csv.

If you want to pass in a path object, pandas accepts any `os.PathLike`.

By file-like object, we refer to objects with a `read()` method, such as a file handler (e.g. via builtin `open` function) or `StringIO`.

sep [str, default ','] 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'`.

delimiter [str, default `None`] Alias for sep.

header [int, list of int, 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 integers that specify row locations for a multi-index 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.

names [array-like, optional] List of column names to use. If the file contains a header row, then you should explicitly pass `header=0` to override the column names. Duplicates in this list are not allowed.

index_col [int, str, sequence of int / str, or False, 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.

usecols [list-like or callable, optional] 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). 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. An example of a valid callable argument would be `lambda x: x.upper() in ['AAA', 'BBB', 'DDD']`. Using this parameter results in much faster parsing time and lower memory usage.

squeeze [bool, default False] If the parsed data only contains one column then return a Series.

prefix [str, optional] Prefix to add to column numbers when no header, e.g. 'X' for X0, X1, …

mangle_dupe_cols [bool, 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.

dtype [Type name or dict of column -> type, optional] Data type for data or columns. E.g. {'a': np.float64, 'b': np.int32, 'c': 'Int64'} Use *str* or *object* together with suitable *na_values* settings to preserve and not interpret dtype. If converters are specified, they will be applied INSTEAD of dtype conversion.

engine [{'c', 'python'}, optional] Parser engine to use. The C engine is faster while the python engine is currently more feature-complete.

converters [dict, optional] Dict of functions for converting values in certain columns. Keys can either be integers or column labels.

true_values [list, optional] Values to consider as True.

false_values [list, optional] Values to consider as False.

skipinitialspace [bool, default False] Skip spaces after delimiter.

skiprows [list-like, int or callable, optional] 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. An example of a valid callable argument would be `lambda x: x in [0, 2]`.

skipfooter [int, default 0] Number of lines at bottom of file to skip (Unsupported with engine='c').

nrows [int, optional] Number of rows of file to read. Useful for reading pieces of large files.

na_values [scalar, str, list-like, or dict, optional] Additional strings to recognize as NA/NaN. If dict passed, specific per-column NA values. By default the following values are interpreted as NaN: '', '#N/A', '#N/A N/A', '#NA', '-1.#IND', '-1.#QNAN', '-NaN', '-nan', '1.#IND', '1.#QNAN', '<NA>', 'N/A', 'NA', 'NULL', 'NaN', 'n/a', 'nan', 'null'.

keep_default_na [bool, 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_filter [bool, 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.

verbose [bool, default False] Indicate number of NA values placed in non-numeric columns.

skip_blank_lines [bool, default True] If True, skip over blank lines rather than interpreting as NaN values.

parse_dates [bool or list of int or names or list of lists or dict, default False] The behavior is as follows:

- boolean. If True -> try parsing the index.
- list of int or names. e.g. If [1, 2, 3] -> try parsing columns 1, 2, 3 each as a separate date column.
- list of lists. e.g. If [[1, 3]] -> combine columns 1 and 3 and parse as a single date column.
- dict, e.g. {'foo' : [1, 3]} -> parse columns 1, 3 as date and call result 'foo'

If a column or index cannot be represented as an array of datetimes, say because of an unparseable value or a mixture of timezones, the column or index will be returned unaltered as an object data type. For non-standard datetime parsing, use `pd.to_datetime` after `pd.read_csv`. To parse an index or column with a mixture of timezones, specify `date_parser` to be a partially-applied *`pandas.to_datetime()`* with `utc=True`. See *Parsing a CSV with mixed timezones* for more.

Note: A fast-path exists for iso8601-formatted dates.

infer_datetime_format [bool, default False] If True and *parse_dates* is enabled, pandas will attempt to infer the format of the datetime strings in the columns, and if it can be inferred, switch to a faster method of parsing them. In some cases this can increase the parsing speed by 5-10x.

keep_date_col [bool, default False] If True and *parse_dates* specifies combining multiple columns then keep the original columns.

date_parser [function, optional] 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.

dayfirst [bool, default False] DD/MM format dates, international and European format.

cache_dates [bool, 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.

iterator [bool, default False] Return TextFileReader object for iteration or getting chunks with `get_chunk()`.

chunksize [int, optional] Return TextFileReader object for iteration. See the IO Tools docs for more information on `iterator` and `chunksize`.

compression [{'infer', 'gzip', 'bz2', 'zip', 'xz', None}, default 'infer'] For on-the-fly decompression of on-disk data. If 'infer' and *filepath_or_buffer* is path-like, then detect compression from the following extensions: '.gz', '.bz2', '.zip', or '.xz' (otherwise no decompression). If using 'zip', the ZIP file must contain only one data file to be read in. Set to None for no decompression.

thousands [str, optional] Thousands separator.

decimal [str, default '.'] Character to recognize as decimal point (e.g. use ',' for European data).

lineterminator [str (length 1), optional] Character to break file into lines. Only valid with C parser.

quotechar [str (length 1), optional] The character used to denote the start and end of a quoted item. Quoted items can include the delimiter and it will be ignored.

quoting [int 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).

doublequote [bool, 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.

escapechar [str (length 1), optional] One-character string used to escape other characters.

comment [str, optional] 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.

encoding [str, optional] Encoding to use for UTF when reading/writing (ex. 'utf-8'). List of Python standard encodings .

dialect [str or csv.Dialect, optional] 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_bad_lines [bool, default True] 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.

warn_bad_lines [bool, default True] If error_bad_lines is False, and warn_bad_lines is True, a warning for each "bad line" will be output.

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

low_memory [bool, 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_map [bool, 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.

float_precision [str, optional] 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.

Returns

DataFrame or TextParser A comma-separated values (csv) file is returned as two-dimensional data structure with labeled axes.

See also:

`to_csv` Write DataFrame to a comma-separated values (csv) file.

read_csv Read a comma-separated values (csv) file into DataFrame.

read_fwf Read a table of fixed-width formatted lines into DataFrame.

Examples

```
>>> pd.read_csv('data.csv')
```

pandas.read_fwf

`pandas.`**`read_fwf`**(*filepath_or_buffer: Union[str, pathlib.Path, IO[~ AnyStr]], colspecs='infer', widths=None, infer_nrows=100, **kwds*)

Read a table of fixed-width formatted lines into DataFrame.

Also supports optionally iterating or breaking of the file into chunks.

Additional help can be found in the online docs for IO Tools.

Parameters

filepath_or_buffer [str, path object or file-like object] Any valid string path is acceptable. The string could be a URL. Valid URL schemes include http, ftp, s3, and file. For file URLs, a host is expected. A local file could be: `file://localhost/path/to/table.csv`.

If you want to pass in a path object, pandas accepts any `os.PathLike`.

By file-like object, we refer to objects with a `read()` method, such as a file handler (e.g. via builtin `open` function) or `StringIO`.

colspecs [list of tuple (int, int) or 'infer'. optional] A list of 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 which are not being skipped via skiprows (default='infer').

widths [list of int, optional] A list of field widths which can be used instead of 'colspecs' if the intervals are contiguous.

infer_nrows [int, default 100] The number of rows to consider when letting the parser determine the *colspecs*.

New in version 0.24.0.

****kwds** [optional] Optional keyword arguments can be passed to `TextFileReader`.

Returns

DataFrame or TextParser A comma-separated values (csv) file is returned as two-dimensional data structure with labeled axes.

See also:

`to_csv` Write DataFrame to a comma-separated values (csv) file.

read_csv Read a comma-separated values (csv) file into DataFrame.

Examples

```
>>> pd.read_fwf('data.csv')
```

3.1.3 Clipboard

read_clipboard([sep])	Read text from clipboard and pass to read_csv.

pandas.read_clipboard

pandas.**read_clipboard**(*sep='\\s+'*, ***kwargs*)

Read text from clipboard and pass to read_csv.

Parameters

sep [str, default 's+'] A string or regex delimiter. The default of 's+' denotes one or more whitespace characters.

****kwargs** See read_csv for the full argument list.

Returns

DataFrame A parsed DataFrame object.

3.1.4 Excel

read_excel(io[, sheet_name, header, names, . . .])	Read an Excel file into a pandas DataFrame.
ExcelFile.parse(self[, sheet_name, header, . . .])	Parse specified sheet(s) into a DataFrame.

pandas.read_excel

pandas.**read_excel**(*io*, *sheet_name=0*, *header=0*, *names=None*, *index_col=None*, *usecols=None*, *squeeze=False*, *dtype=None*, *engine=None*, *converters=None*, *true_values=None*, *false_values=None*, *skiprows=None*, *nrows=None*, *na_values=None*, *keep_default_na=True*, *verbose=False*, *parse_dates=False*, *date_parser=None*, *thousands=None*, *comment=None*, *skipfooter=0*, *convert_float=True*, *mangle_dupe_cols=True*, ***kwds*)

Read an Excel file into a pandas DataFrame.

Supports *xls*, *xlsx*, *xlsm*, *xlsb*, and *odf* file extensions read from a local filesystem or URL. Supports an option to read a single sheet or a list of sheets.

Parameters

io [str, bytes, ExcelFile, xlrd.Book, path object, or file-like object] Any valid string path is acceptable. The string could be a URL. Valid URL schemes include http, ftp, s3, and file. For file URLs, a host is expected. A local file could be: `file://localhost/path/to/table.xlsx`.

If you want to pass in a path object, pandas accepts any `os.PathLike`.

By file-like object, we refer to objects with a `read()` method, such as a file handler (e.g. via builtin `open` function) or `StringIO`.