pandas.read_csv

pandas.read_csv(*filepath_or_buffer, *, sep=<no_default>, delimiter=None, header='infer', names=<no_default>, index_col=None, usecols=None, 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=<no_default>, skip_blank_lines=True, parse_dates=None, infer_datetime_format=<no_default>, keep_date_col=<no_default>, date_parser=<no_default>, date_format=None, dayfirst=False, cache_dates=True, iterator=False, chunksize=None, compression='infer', thousands=None, decimal='.', lineterminator=None, quotechar='"', quoting=0, doublequote=True, escapechar=None, comment=None, encoding=None, encoding_errors='strict', dialect=None, on_bad_lines='error', delim_whitespace=<no_default>, low_memory=True, memory_map=False, float_precision=None, storage_options=None, dtype_backend=<no_default>*) [source]

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, gs, 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 handle (e.g. via builtin `open` function) or `StringIO`.

Skip to main content

Character or regex pattern to treat as the delimiter. If `sep=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 from only the first valid row of the file 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, optional*

Alias for `sep`.

header : *int, Sequence of int, 'infer' or None, default 'infer'*

Row number(s) containing column labels and marking the start of the data (zero-indexed). 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 to `names` 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 `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.

names : *Sequence of Hashable, optional*

Sequence of column labels to apply. 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 : *Hashable, Sequence of Hashable or False, optional*

Column(s) to use as row label(s), denoted either by column labels or column indices. If a sequence of labels or indices is given, `MultiIndex` will be formed for the row labels.

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 : *Sequence of Hashable or Callable, optional*

Subset of columns to select, denoted either by column labels or column indices. If list-like, all elements must either be positional (i.e. integer indices into the document

Skip to main content

`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`. 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.

dtype : *dtype or dict of {Hashable : dtype}, optional*

Data type(s) to apply to either the whole dataset or individual 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.

> ***Added in version 1.5.0:*** Support for `defaultdict` was added. Specify a `defaultdict` as input where the default determines the `dtype` of the columns which are not explicitly listed.

engine : *{'c', 'python', 'pyarrow'}, optional*

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.

> ***Added 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.

converters : *dict of {Hashable : Callable}, optional*

Functions for converting values in specified columns. Keys can either be column labels or column indices

true_values : *list, optional*

Values to consider as `True` in addition to case-insensitive variants of 'True'.

false_values : *list, optional*

Values to consider as `False` in addition to case-insensitive variants of 'False'.

skipinitialspace : *bool, default False*

Skip spaces after delimiter.

skiprows : *int, list of 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 : *Hashable, Iterable of Hashable or dict of {Hashable : Iterable}, 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", "None", "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`.

Skip to main content

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 `NA` values, 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.

> **Deprecated since version 2.2.0.**

skip_blank_lines : *bool, default True*

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

parse_dates : *bool, list of Hashable, list of lists or dict of {Hashable : list}, default False*

The behavior is as follows:

- `bool`. If `True` -> try parsing the index. Note: Automatically set to `True` if `date_format` or `date_parser` arguments have been passed.
- `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 `list`. e.g. If `[[1, 3]]` -> combine columns 1 and 3 and parse as a single date column. Values are joined with a space before parsing.
- `dict`, e.g. `{'foo' : [1, 3]}` -> parse columns 1, 3 as date and call result 'foo'. Values are joined with a space before parsing.

If a column or index cannot be represented as an array of `datetime`, say because of an unparsable 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 `to_datetime()` after `read_csv()`.

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

Deprecated since version 2.0.0: A strict version of this argument is now the default, passing it has no effect.

keep_date_col : *bool, default False*

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

date_parser : *Callable, 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.

Deprecated since version 2.0.0: Use `date_format` instead, or read in as `object` and then apply `to_datetime()` as-needed.

date_format : *str or dict of column -> format, optional*

Format to use for parsing dates when used in conjunction with `parse_dates`. The strftime to parse time, e.g. `"%d/%m/%Y"`. See strftime documentation for more information on choices, though note that `"%f"` will parse all the way up to nanoseconds. You can also pass:

- **"ISO8601", to parse any ISO8601**

  time string (not necessarily in exactly the same format);

- **"mixed", to infer the format for each element individually. This is risky,**

  and you should probably use it along with *dayfirst*.

Added in version 2.0.0.

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.

iterator : *bool, default False*

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

chunksize : *int, optional*

Number of lines to read from the file per chunk. Passing a value will cause the function to return a `TextFileReader` object for iteration. See the IO Tools docs for more information on `iterator` and `chunksize`.

compression : *str or dict, 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', '.xz', '.zst', '.tar', '.tar.gz', '.tar.xz' or '.tar.bz2' (otherwise no compression). If using 'zip' or 'tar', 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'`, `'xz'`, `'tar'`} and other key-value pairs are forwarded to `zipfile.ZipFile`, `gzip.GzipFile`, `bz2.BZ2File`, `zstandard.ZstdDecompressor`, `lzma.LZMAFile` or `tarfile.TarFile`, respectively. As an example, the following could be passed for Zstandard decompression using a custom compression dictionary: `compression={'method': 'zstd', 'dict_data': my_compression_dict}`.

> ***Added in version 1.5.0:*** Added support for *.tar* files.

> ***Changed in version 1.4.0:*** Zstandard support.

thousands : *str (length 1), optional*

Character acting as the thousands separator in numerical values.

decimal : *str (length 1), default '.'*

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

lineterminator : *str (length 1), optional*

Character used to denote a line break. Only valid with C parser

Skip to main content

quotechar : *str (length 1), optional*

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

quoting : *{0 or csv.QUOTE_MINIMAL, 1 or csv.QUOTE_ALL, 2 or csv.QUOTE_NONNUMERIC, 3 or csv.QUOTE_NONE}, default csv.QUOTE_MINIMAL*

Control field quoting behavior per `csv.QUOTE_*` constants. Default is `csv.QUOTE_MINIMAL` (i.e., 0) which implies that only fields containing special characters are quoted (e.g., characters defined in `quotechar`, `delimiter`, or `lineterminator`.

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*

Character used to escape other characters.

comment : *str (length 1), optional*

Character indicating that the 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, default 'utf-8'*

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

encoding_errors : *str, optional, default 'strict'*

How encoding errors are treated. List of possible values .

> *Added in version 1.3.0.*

dialect : *str or csv.Dialect, optional*

If provided, this parameter will override values (default or not) for the following

Skip to main content

and `quoting`. If it is necessary to override values, a `ParserWarning` will be issued. See `csv.Dialect` documentation for more details.

on_bad_lines : *{'error', 'warn', 'skip'} or Callable, default 'error'*

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

- `'error'`, raise an Exception when a bad line is encountered.
- `'warn'`, raise a warning when a bad line is encountered and skip that line.
- `'skip'`, skip bad lines without raising or warning when they are encountered.

> ***Added in version 1.3.0.***

> ***Added in version 1.4.0:***
>
> - Callable, function with signature `(bad_line: list[str]) -> list[str] | None` that will process a single bad line. `bad_line` is a list of strings split by the `sep`. If the function returns `None`, the bad line will be ignored. If the function returns a new `list` of strings with more elements than expected, a `ParserWarning` will be emitted while dropping extra elements. Only supported when `engine='python'`

> ***Changed in version 2.2.0:***
>
> - Callable, function with signature as described in pyarrow documentation when `engine='pyarrow'`

delim_whitespace : *bool, default False*

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

> ***Deprecated since version 2.2.0:*** Use `sep="\s+"` instead.

low_memory : *bool, default True*

Internally process the file in chunks, resulting in lower memory use while parsing, but

Skip to main content

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 : *{'high', 'legacy', 'round_trip'}, optional*

Specifies which converter the C engine should use for floating-point values. The options are `None` or `'high'` for the ordinary converter, `'legacy'` for the original lower precision pandas converter, and `'round_trip'` for the round-trip converter.

storage_options : *dict, optional*

Extra options that make sense for a particular storage connection, e.g. host, port, username, password, etc. For HTTP(S) URLs the key-value pairs are forwarded to `urllib.request.Request` as header options. For other URLs (e.g. starting with "s3://", and "gcs://") the key-value pairs are forwarded to `fsspec.open`. Please see `fsspec` and `urllib` for more details, and for more examples on storage options refer here.

dtype_backend : *{'numpy_nullable', 'pyarrow'}, default 'numpy_nullable'*

Back-end data type applied to the resultant `DataFrame` (still experimental). Behaviour is as follows:

- `"numpy_nullable"`: returns nullable-dtype-backed `DataFrame` (default).
- `"pyarrow"`: returns pyarrow-backed nullable `ArrowDtype` DataFrame.

Added in version 2.0.

Returns:

DataFrame or TextFileReader

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

See also

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

`read_table`
: Read general delimited file into DataFrame.

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