# pandas.read_csv

pandas.read_csv(*filepath_or_buffer*, *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*, *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*, *iterator=False*, *chunksize=None*, *compression='infer'*, *thousands=None*, *decimal=b'.'*, *lineterminator=None*, *quotechar='"'*, *quoting=0*, *escapechar=None*, *comment=None*, *encoding=None*, *dialect=None*, *tupleize_cols=False*, *error_bad_lines=True*, *warn_bad_lines=True*, *skipfooter=0*, *skip_footer=0*, *doublequote=True*, *delim_whitespace=False*, *as_recarray=False*, *compact_ints=False*, *use_unsigned=False*, *low_memory=True*, *buffer_lines=None*, *memory_map=False*, *float_precision=None*) [source]

Read CSV (comma-separated) 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, pathlib.Path, py._path.local.LocalPath or any object with a read() method (such as a file handle or 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.csv

**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 automatically. 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`

Alternative argument name for sep.

**delim_whitespace** *: boolean, 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.

*New in version 0.18.1:* support for the Python parser.

**header** *: int or list of ints, default ‘infer’*

Row number(s) to use as the column names, and the start of the data. Default behavior is as if set to 0 if no `names` passed, otherwise `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, 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 unless mangle_dupe_cols=True, which is the default.

**index_col** *: int or sequence or False, default None*

Column to use as the row labels of the DataFrame. If a sequence is given, a MultiIndex is used. If you have a malformed file with delimiters at the end of each line, you might consider index_col=False to force pandas to _not_ use the first column as the index (row names)

**usecols** *: array-like or callable, default None*

Return a subset of the columns. If array-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 array-like *usecols* parameter would be [0, 1, 2] or [‘foo’, ‘bar’, ‘baz’].
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.

**as_recarray** *: boolean, default False*

DEPRECATED: this argument will be removed in a future version. Please call *pd.read_csv(...).to_records()* instead.
Return a NumPy recarray instead of a DataFrame after parsing the data. If set to True, this option takes precedence over the *squeeze* parameter. In addition, as row indices are not available in such a format, the *index_col* parameter will be ignored.

**squeeze** *: boolean, default False*

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

**prefix** *: str, default None*

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

**mangle_dupe_cols** *: boolean, default True*

Duplicate columns will be specified as ‘X.0’...’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, default None*

Data type for data or columns. E.g. {‘a’: np.float64, ‘b’: np.int32} Use *str* or *object* 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, default None*

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

**true_values** *: list, default None*

Values to consider as True

**false_values** *: list, default None*
Values to consider as False

**skipinitialspace** *: boolean, default False*
Skip spaces after delimiter.

**skiprows** *: list-like or integer or callable, 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. 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')

**skip_footer** *: int, default 0*
DEPRECATED: use the *skipfooter* parameter instead, as they are identical

**nrows** *: int, default None*
Number of rows of file to read. Useful for reading pieces of large files

**na_values** *: scalar, str, list-like, or dict, default None*
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', 'N/A', 'NA', 'NULL', 'NaN', 'nan'`.

**keep_default_na** *: bool, default True*
If na_values are specified and keep_default_na is False the default NaN values are overridden, otherwise they're appended to.

**na_filter** *: boolean, 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** *: boolean, default False*
Indicate number of NA values placed in non-numeric columns

**skip_blank_lines** *: boolean, default True*
If True, skip over blank lines rather than interpreting as NaN values

**parse_dates** *: boolean or list of ints or names or list of lists or dict, default False*

- boolean. If True -> try parsing the index.
- list of ints 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 contains an unparseable date, the entire 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`
Note: A fast-path exists for iso8601-formatted dates.

**infer_datetime_format** *: boolean, 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** *: boolean, default False*

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

**date_parser** *: function, 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.

**dayfirst** *: boolean, default False*

DD/MM format dates, international and European format

**iterator** *: boolean, default False*

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

**chunksize** *: int, default None*

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', then use gzip, bz2, zip or xz if filepath_or_buffer is a string ending in '.gz', '.bz2', '.zip', or 'xz', 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.

*New in version 0.18.1:* support for 'zip' and 'xz' compression.

**thousands** *: str, default None*

Thousands separator

**decimal** *: str, default '.'*

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

**float_precision** *: string, 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.

**lineterminator** *: str (length 1), default None*

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** *: boolean, 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), default None*

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

**comment** *: str, 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 '#emptyna,b,cn1,2,3' with *header=0* will result in 'a,b,c' being treated as the header.

**encoding** *: str, default None*

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

**dialect** *: str 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.

**tupleize_cols** *: boolean, default False*

Leave a list of tuples on columns as is (default is to convert to a Multi Index on the columns)

**error_bad_lines** *: boolean, 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** *: boolean, default True*

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

**low_memory** *: boolean, 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)

**buffer_lines** *: int, default None*

DEPRECATED: this argument will be removed in a future version because its value is not respected by the parser

**compact_ints** *: boolean, default False*

DEPRECATED: this argument will be removed in a future version
If compact_ints is True, then for any column that is of integer dtype, the parser will attempt to cast it as the smallest integer dtype possible, either signed or unsigned depending on the specification from the *use_unsigned* parameter.

**use_unsigned** *: boolean, default False*

DEPRECATED: this argument will be removed in a future version
If integer columns are being compacted (i.e. *compact_ints=True*), specify whether the column should be compacted to the smallest signed or unsigned integer dtype.

**memory_map** *: boolean, 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.

**Returns:** **result** *: DataFrame or TextParser*