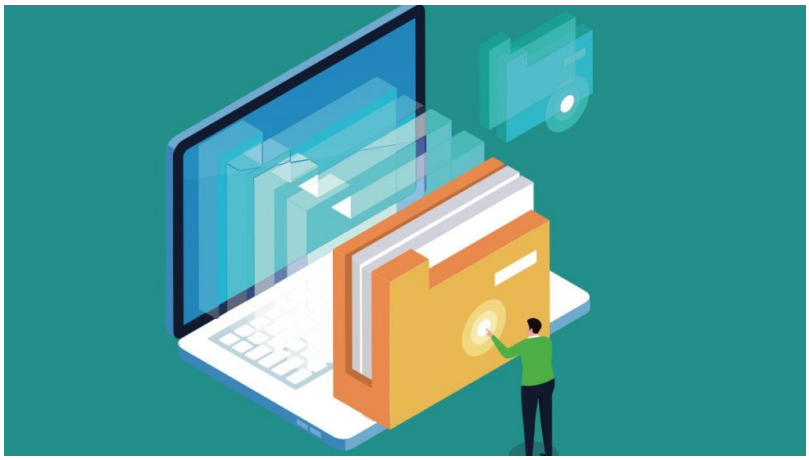
Reading CSV and TXT files



Rather than creating `Series` or `DataFrames` strutures from scratch, or even from Python core sequences or `ndarrays` , the most typical use of **pandas** is based on the loading of information from files or sources of information for further exploration, transformation and analysis.

In this lecture we'll learn how to read comma-separated values files (.csv) and raw text files (.txt) into pandas `DataFrame` s.

Hands on!

In [ ]:

```
import pandas as pd
```

Reading data with Python

As we saw on previous courses we can read data simply using Python.

When you want to work with a file, the first thing to do is to open it. This is done by invoking the `open()` built-in function.

`open()` has a single required argument that is the path to the file and has a single return, the file object.

The `with` statement automatically takes care of closing the file once it leaves the `with` block, even in cases of error.

In [ ]:

```
with open('btc-market-price.csv', 'r') as fp:
    print(fp)
```

Once the file is opened, we can read its content as follows:

In [ ]:

```
with open('btc-market-price.csv', 'r') as fp:
    for index, line in enumerate(fp.readlines()):
        # read just the first 10 lines
        if (index < 10):
            print(index, line)
```

How can we process the data read from the file using pure Python? It involves a lot of manual work, for example, splitting the values by the correct separator:

In [ ]:

```
with open('btc-market-price.csv', 'r') as fp:
    for index, line in enumerate(fp.readlines()):
        # read just the first 10 lines
        if (index < 10):
            timestamp, price = line.split(',')
            print(f"{timestamp}: ${price}")
```

But what happens if the separator is unknown, like in the file `exam_review.csv` :

In [ ]:

```
!head exam_review.csv
```

In this case, the separator is not a *comma*, but the `>` sign. It's still a "CSV", although not technically separated by commas.

The `csv` module

Python includes the builtin module `csv` that helps a little bit more with the process of reading CSVs:

In [ ]:

```
import csv
```

In [ ]:

```
with open('btc-market-price.csv', 'r') as fp:
    reader = csv.reader(fp)
    for index, (timestamp, price) in enumerate(reader):
        # read just the first 10 lines
        if (index < 10):
            print(f"{timestamp}: ${price}")
```

The `csv` modules takes care of splitting the file using a given separator (called `delimiter`) and creating an iterator for us.

In [ ]:

```
with open('exam_review.csv', 'r') as fp:
    reader = csv.reader(fp, delimiter='>')  # special delimiter
    next(reader)  # skipping header
    for index, values in enumerate(reader):
        if not values:
            continue  # skip empty lines
        fname, lname, age, math, french = values
        print(f"{fname} {lname} (age {age}) got {math} in Math and {french} in F
rench")
```

Reading data with Pandas

Probably one of the most recurrent types of work for data analysis: public data sources, logs, historical information tables, exports from databases. So the pandas library offers us functions to read and write files in multiple formats like CSV, JSON, XML and Excel's XLSX, all of them creating a `DataFrame` with the information read from the file.

We'll learn how to read different type of data including:

- CSV files (.csv)
- Raw text files (.txt)
- JSON data from a file and from an API
- Data from a SQL query over a database

There are many other available reading functions as the following table shows:

Format Type	Data Description	Reader	Writer
text	CSV	read_csv	to_csv
text	JSON	read_json	to_json
text	HTML	read_html	to_html
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	Msgpack	read_msgpack	to_msgpack
binary	Stata	read_stata	to_stata
binary	SAS	read_sas	
binary	Python Pickle Format	read_pickle	to_pickle
SQL	SQL	read_sql	to_sql
SQL	Google Big Query	read_gbq	to_gbq

The `read_csv` method

The first method we'll learn is **read_csv**, that let us read comma-separated values (CSV) files and raw text (TXT) files into a `DataFrame` .

The `read_csv` function is extremely powerful and you can specify a very broad set of parameters at import time that allow us to accurately configure how the data will be read and parsed by specifying the correct structure, enconding and other details. The most common parameters are as follows:

- `filepath` : Path of the file to be read.
- `sep` : Character(s) that are used as a field separator in the file.
- `header` : Index of the row containing the names of the columns (None if none).
- `index_col` : Index of the column or sequence of indexes that should be used as index of rows of the data.
- `names` : Sequence containing the names of the columns (used together with header = None).
- `skiprows` : Number of rows or sequence of row indexes to ignore in the load.
- `na_values` : Sequence of values that, if found in the file, should be treated as NaN.
- `dtype` : Dictionary in which the keys will be column names and the values will be types of NumPy to which their content must be converted.
- `parse_dates` : Flag that indicates if Python should try to parse data with a format similar to dates as dates. You can enter a list of column names that must be joined for the parsing as a date.
- `date_parser` : Function to use to try to parse dates.
- `nrows` : Number of rows to read from the beginning of the file.
- `skip_footer` : Number of rows to ignore at the end of the file.
- `encoding` : Encoding to be expected from the file read.
- `squeeze` : Flag that indicates that if the data read only contains one column the result is a Series instead of a DataFrame.
- `thousands` : Character to use to detect the thousands separator.
- `decimal` : Character to use to detect the decimal separator.
- `skip_blank_lines` : Flag that indicates whether blank lines should be ignored.

> Full `read_csv` documentation can be found here: https://pandas.pydata.org/pandas-docs/stable/reference/api/pandas.read_csv.html (https://pandas.pydata.org/pandas-docs/stable/reference/api/pandas.read_csv.html).

In this case we'll try to read our `btc-market-price.csv` CSV file using different parameters to parse it correctly.

This file contains records of the mean price of Bitcoin per date.

Reading our first CSV file

Everytime we call `read_csv` method, we'll need to pass an explicit `filepath` parameter indicating the path where our CSV file is.

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` .

For example we can use `read_csv` method to load data directly from an URL:

In [ ]:

```
csv_url = "https://raw.githubusercontent.com/datasets/gdp/master/data/gdp.csv"

pd.read_csv(csv_url).head()
```

Or just use a local file:

In [ ]:

```
df = pd.read_csv('btc-market-price.csv')

df.head()
```

In this case we let pandas infer everything related to our data, but in most of the cases we'll need to explicitly tell pandas how we want our data to be loaded. To do that we use parameters.

Let's see how theses parameters work.

First row behaviour with `header` parameter

The CSV file we're reading has only two columns: `Timestamp` and `Price` . It doesn't have a header. Pandas automatically assigned the first row of data as headers, which is incorrect. We can overwrite this behavior with the `header` parameter.

In [ ]:

```
df = pd.read_csv('btc-market-price.csv',
                 header=None)
```

In [ ]:

```
df.head()
```

Missing values with `na_values` parameter

We can define a `na_values` parameter with the values we want to be recognized as NA/NaN. In this case empty strings `''`, `?` and `-` will be recognized as null values.

In [ ]:

```
df = pd.read_csv('btc-market-price.csv',
                 header=None,
                 na_values=['', '?', '-'])
```

In [ ]:

```
df.head()
```

Column names using `names` parameter

We'll add that columns names using the `names` parameter.

In [ ]:

```
df = pd.read_csv('btc-market-price.csv',
                 header=None,
                 na_values=['', '?', '-'],
                 names=['Timestamp', 'Price'])
```

In [ ]:

```
df.head()
```

In [ ]:

```
df.info()
```

Column types using `dtype` parameter

Without using the `dtype` parameter pandas will try to figure it out the type of each column automatically. We can use `dtype` parameter to force pandas to use certain dtype.

In this case we'll force the `Price` column to be `float`.

In [ ]:

```
df = pd.read_csv('btc-market-price.csv',
                 header=None,
                 na_values=['', '?', '-'],
                 names=['Timestamp', 'Price'],
                 dtype={'Price': 'float'})
```

In [ ]:

```
df.head()
```

In [ ]:

```
df.dtypes
```

The `Timestamp` column was interpreted as a regular string (`object` in pandas notation), we can parse it manually using a vectorized operation as we saw on previous courses.

We'll parse `Timestamp` column to `Datetime` objects using `to_datetime` method:

In [ ]:

```
pd.to_datetime(df['Timestamp']).head()
```

In [ ]:

```
df['Timestamp'] = pd.to_datetime(df['Timestamp'])
```

In [ ]:

```
df.head()
```

In [ ]:

```
df.dtypes
```

Date parser using `parse_dates` parameter

Another way of dealing with `Datetime` objects is using `parse_dates` parameter with the position of the columns with dates.

In [ ]:

```
df = pd.read_csv('btc-market-price.csv',
                 header=None,
                 na_values=['', '?', '-'],
                 names=['Timestamp', 'Price'],
                 dtype={'Price': 'float'},
                 parse_dates=[0])
```

In [ ]:

```
df.head()
```

In [ ]:

```
df.dtypes
```

Adding index to our data using `index_col` parameter

By default, pandas will automatically assign a numeric autoincremental index or row label starting with zero. You may want to leave the default index as such if your data doesn't have a column with unique values that can serve as a better index. In case there is a column that you feel would serve as a better index, you can override the default behavior by setting `index_col` property to a column. It takes a numeric value representing the index or a string of the column name for setting a single column as index or a list of numeric values or strings for creating a multi-index.

In our data, we are choosing the first column, `Timestamp` , as index (index=0) by passing zero to the `index_col` argument.

In [ ]:

```
df = pd.read_csv('btc-market-price.csv',
                 header=None,
                 na_values=['', '?', '-'],
                 names=['Timestamp', 'Price'],
                 dtype={'Price': 'float'},
                 parse_dates=[0],
                 index_col=[0])
```

In [ ]:

```
df.head()
```

In [ ]:

```
df.dtypes
```

A more challenging parsing

Now we'll read another CSV file. This file has the following columns:

- `first_name`
- `last_name`
- `age`
- `math_score`
- `french_score`
- `next_test_date`

Let's read it and see how it looks like.

In [ ]:

```
exam_df = pd.read_csv('exam_review.csv')
```

In [ ]:

```
exam_df
```

Custom data delimiters using `sep` parameter

We can define which delimiter to use by using the `sep` parameter. If we don't use the `sep` parameter, pandas will automatically detect the separator.

In most of the CSV files separator will be comma (`,`) and will be automatically detected. But we can find files with other separators like semicolon (`;`), tabs (`\t` , specially on TSV files), whitespaces or any other special character.

In this case the separator is a `>` character

In [ ]:

```
exam_df = pd.read_csv('exam_review.csv',
                      sep='>')
```

In [ ]:

```
exam_df
```

Custom data encoding

Files are stored using different "encodings". You've probably heard about ASCII, UTF-8, latin1, etc.

While reading data custom encoding can be defined with the `encoding` parameter.

- `encoding='UTF-8'` : will be used if data is UTF-8 encoded.
- `encoding='iso-8859-1'` : will be used if data is ISO/IEC 8859-1 ("extended ASCII") encoded.

In our case we don't need a custom enconding as data is properly loaded.

Custom numeric `decimal` and `thousands` character

The decimal and thousands characters could change between datasets. If we have a column containing a comma (`,`) to indicate the decimal or thousands place, then this column would be considered a string and not numeric.

In [ ]:

```
exam_df = pd.read_csv('exam_review.csv',
                      sep='>')
```

In [ ]:

```
exam_df
```

In [ ]:

```
exam_df[['math_score', 'french_score']].dtypes
```

To solve that, ensuring such columns are interpreted as integer values, we'll need to use the `decimal` and/or `thousands` parameters to indicate correct decimal and/or thousands indicators.

In [ ]:
```
exam_df = pd.read_csv('exam_review.csv',
                      sep='>',
                      decimal=',')
```

In [ ]:
```
exam_df
```

In [ ]:
```
exam_df[['math_score', 'french_score']].dtypes
```

Let's see what happens with the `thousands` parameter:

In [ ]:
```
pd.read_csv('exam_review.csv',
            sep='>',
            thousands=',')
```

Excluding specific rows

We can use the `skiprows` to:

- Exclude reading specified number of rows from the beginning of a file, by passing an integer argument. **This removes the header too**.
- Skip reading specific row indices from a file, by passing a list containing row indices to skip.

In [ ]:
```
exam_df = pd.read_csv('exam_review.csv',
                      sep='>',
                      decimal=',')
```

In [ ]:
```
exam_df
```

To skip reading the first 2 rows from this file, we can use `skiprows=2` :

In [ ]:
```
pd.read_csv('exam_review.csv',
            sep='>',
            skiprows=2)
```

As the header is considered as the first row, to skip reading data rows 1 and 3, we can use `skiprows=[1,3]` :

In [ ]:

```
exam_df = pd.read_csv('exam_review.csv',
                      sep='>',
                      decimal=',',
                      skiprows=[1,3])
```

In [ ]:

```
exam_df
```

Get rid of blank lines

The `skip_blank_lines` parameter is set to `True` so blank lines are skipped while we read files.

If we set this parameter to `False` , then every blank line will be loaded with `NaN` values into the `DataFrame` .

In [ ]:

```
pd.read_csv('exam_review.csv',
            sep='>',
            skip_blank_lines=False)
```

Loading specific columns

We can use the `usecols` parameter when we want to load just specific columns and not all of them.

Performance wise, it is better because instead of loading an entire dataframe into memory and then deleting the not required columns, we can select the columns that we'll need, while loading the dataset itself.

As a parameter to `usecols` , you can pass either a list of strings corresponding to the column names or a list of integers corresponding to column index.

In [ ]:

```
pd.read_csv('exam_review.csv',
            usecols=['first_name', 'last_name', 'age'],
            sep='>')
```

Or using just the column position:

In [ ]:

```
pd.read_csv('exam_review.csv',
            usecols=[0, 1, 2],
            sep='>')
```

Using a `Series` instead of `DataFrame`

If the parsed data only contains one column then we can return a Series by setting the `squeeze` parameter to `True`.

In [ ]:

```
exam_test_1 = pd.read_csv('exam_review.csv',
                          sep='>',
                          usecols=['last_name'])
```

In [ ]:

```
exam_test_1
```

In [ ]:

```
type(exam_test_1)
```

In [ ]:

```
exam_test_2 = pd.read_csv('exam_review.csv',
                          sep='>',
                          usecols=['last_name'],
                          squeeze=True)
```

In [ ]:

```
exam_test_2
```

In [ ]:

```
type(exam_test_2)
```

Save to CSV file

Finally we can also save our `DataFrame` as a CSV file.

In [ ]:

```
exam_df
```

We can simply generate a CSV string from our `DataFrame`:

In [ ]:

```
exam_df.to_csv()
```

Or specify a file path where we want our generated CSV code to be saved:

In [ ]:

```
exam_df.to_csv('out.csv')
```

In [ ]:

```
pd.read_csv('out.csv')
```

In [ ]:

```
exam_df.to_csv('out.csv',
               index=None)
```

In [ ]:

```
pd.read_csv('out.csv')
```