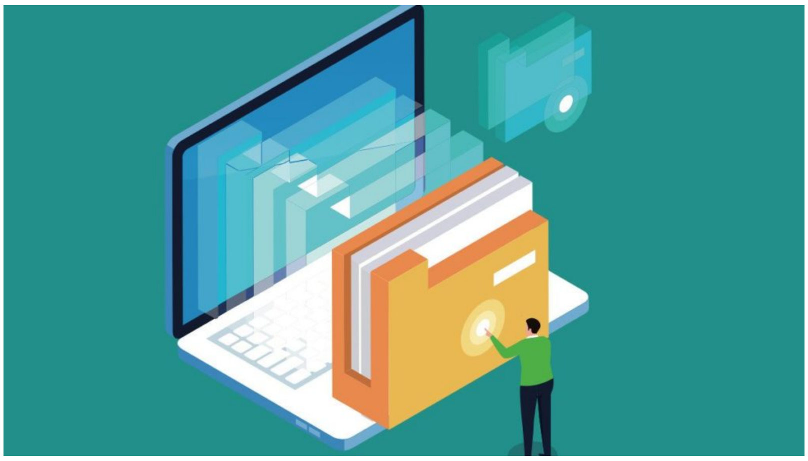
Reading Excel files

In this lecture we'll learn how to read Excel files (.xlsx) and its sheets into a pandas `DataFrame` s, and how to export that `DataFrame` s to different sheets and Excel files using the pandas `ExcelWriter` and `to_excel` methods.



Hands on!

In [ ]:

```
import pandas as pd
```

In [ ]:

```
!head products.xlsx
```

The `read_excel` method

We'll begin with the **read_excel** method, that let us read Excel files into a `DataFrame`.

This method supports both XLS and XLSX file extensions from a local filesystem or URL and has a broad set of parameters to configure how the data will be read and parsed. These parameters are very similar to the parameters we saw on previous lectures where we introduced the `read_csv` method. The most common parameters are as follows:

- `filepath` : Path of the file to be read.
- `sheet_name` : Strings are used for sheet names. Integers are used in zero-indexed sheet positions. Lists of strings/integers are used to request multiple sheets. Specify None to get all sheets.
- `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.
- `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.

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

In this case we'll try to read our `products.xlsx` Excel file.

This file contains records of products with its price, brand, description and merchant information on different sheets.

Reading our first Excel file

Everytime we call `read_excel` method, we'll need to pass an explicit `filepath` parameter indicating the path where our Excel 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.xlsx` .

In [ ]:

```
df = pd.read_excel('products.xlsx')
```

In [ ]:

```
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 Excel file we're reading has the following columns:

- `product_id`
- `price`
- `merchant_id`
- `brand`
- `name`

The first row (0-index) of the data has that column names, so we keep the implicit `header=0` parameter to let Pandas assign this first row as headers. We can overwrite this behavior defining explicitly the `header` parameter.

In [ ]:

```
pd.read_excel('products.xlsx').head()
```

In [ ]:

```
pd.read_excel('products.xlsx',
              header=None).head()
```

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 or a string for setting a single column as index or a list of numeric values for creating a multi-index.

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

In [ ]:

```
df = pd.read_excel('products.xlsx',
                   index_col=[0])
```

In [ ]:

```
df.head()
```

Selecting specific sheets

Excel files quite often have multiple sheets and the ability to read a specific sheet or all of them is very important. To make this easy, the pandas `read_excel` method takes an argument called `sheet_name` that tells pandas which sheet to read in the data from.

For this, you can either use the sheet name or the sheet number. Sheet numbers start with zero. The first sheet will be the one loaded by default. You can change sheet by specifying `sheet_name` parameter.

In [ ]:

```
products = pd.read_excel('products.xlsx',
                         sheet_name='Products',
                         index_col='product_id')
```

In [ ]:

```
products.head()
```

In [ ]:

```
merchants = pd.read_excel('products.xlsx',
                          sheet_name='Merchants',
                          index_col='merchant_id')
```

In [ ]:

```
merchants.head()
```

The `ExcelFile` class

Another approach on reading Excel data is using the `ExcelFile` class for parsing tabular Excel sheets into `DataFrame` objects.

This `ExcelFile` will let us work with sheets easily, and will be faster than the previous `read_excel` method.

In [ ]:

```
excel_file = pd.ExcelFile('products.xlsx')
```

In [ ]:

```
excel_file
```

We can now explore the sheets on that Excel file with `sheet_names` :

In [ ]:

```
excel_file.sheet_names
```

And parse specified sheet(s) into a Pandas' `DataFrame` using ExcelFile's `parse()` method.

Everytime we call `parse()` method, we'll need to pass an explicit `sheet_name` parameter indicating which sheet from the Excel file we want to be parsed. First sheet will be parsed by default.

In [ ]:

```
products = excel_file.parse('Products')
```

In [ ]:

```
products.head()
```

This `parse()` method has all the parameters we saw before on `read_excel()` method, let's try some of them:

In [ ]:

```
products = excel_file.parse(sheet_name='Products',
                            header=0,
                            index_col='product_id')
```

In [ ]:

```
products.head()
```

In [ ]:

```
products.dtypes
```

In [ ]:

```
merchants = excel_file.parse('Merchants',
                             index_col='merchant_id')
```

In [ ]:

```
merchants.head()
```

In [ ]:

```
merchants.dtypes
```

Save to Excel file

Finally we can save our `DataFrame` as a Excel file.

In [ ]:

```
products.head()
```

A fast, simple way to write a single `DataFrame` to an Excel file is to use the `to_excel()` method of the `DataFrame` directly.

Note that it's required to pass a output file path.

> The `OpenPyXL - openpyxl` library should be installed in order to save Excel files. `pip install openpyxl`

In [ ]:

```
products.to_excel('out.xlsx')
```

In [ ]:

```
pd.read_excel('out.xlsx').head()
```

We can specify the sheet name with `sheet_name` parameter:

In [ ]:

```
products.to_excel('out.xlsx',
                  sheet_name='Products')
```

Further calls of `to_excel` with different sheet names will only overwrite the first sheet instead of adding additional sheets.

Also, be aware that by removing the index, we'll lose that column.

In [ ]:

```
products.to_excel('out.xlsx',
                  index=None)
```

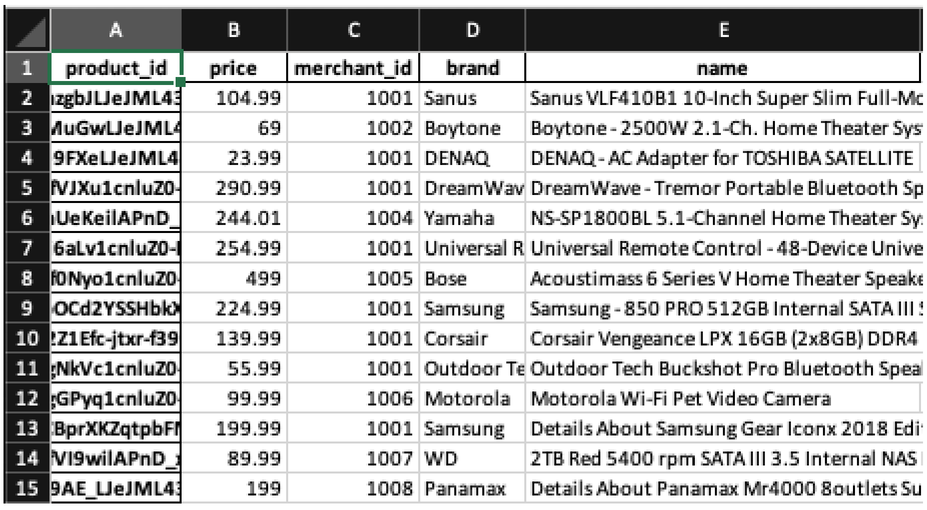
In [ ]:

```
pd.read_excel('out.xlsx').head()
```

Positioning Data with `startrow` and `startcol`

Suppose we wanted to insert the our data into the spreadsheet file in a position somewhere other than the top-left corner.

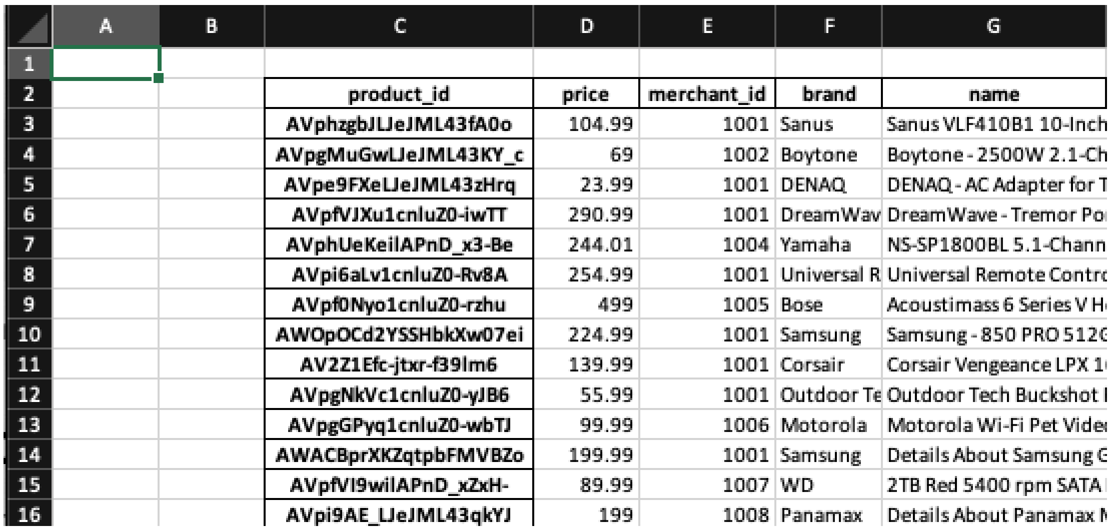
We can shift where the `to_excel` method writes the data by using `startrow` to set the cell after which the first row will be printed, and `startcol` to set which cell after which the first column will be printed.



	A	B	C	D	E
1	product_id	price	merchant_id	brand	name
2	ızgbJLJeJML43	104.99	1001	Sanus	Sanus VLF410B1 10-Inch Super Slim Full-Mc
3	⁄uGwLJeJML4	69	1002	Boytone	Boytone - 2500W 2.1-Ch. Home Theater Sys
4	9FXeLJeJML4	23.99	1001	DENAQ	DENAQ - AC Adapter for TOSHIBA SATELLITE
5	fVJXu1cnluZ0-	290.99	1001	DreamWav	DreamWave - Tremor Portable Bluetooth Sp
6	ıUeKeilAPnD_	244.01	1004	Yamaha	NS-SP1800BL 5.1-Channel Home Theater Sy
7	6aLv1cnluZ0-I	254.99	1001	Universal R	Universal Remote Control - 48-Device Unive
8	f0Nyo1cnluZ0-	499	1005	Bose	Acoustimass 6 Series V Home Theater Speake
9	OCd2YSSHbkX	224.99	1001	Samsung	Samsung - 850 PRO 512GB Internal SATA III
10	2Z1Efc-jtxr-f39	139.99	1001	Corsair	Corsair Vengeance LPX 16GB (2x8GB) DDR4
11	gNkVc1cnluZ0-	55.99	1001	Outdoor Te	Outdoor Tech Buckshot Pro Bluetooth Speal
12	gGPyq1cnluZ0-	99.99	1006	Motorola	Motorola Wi-Fi Pet Video Camera
13	BprXKZqtpbFM	199.99	1001	Samsung	Details About Samsung Gear Iconx 2018 Edi
14	VI9wilAPnD_	89.99	1007	WD	2TB Red 5400 rpm SATA III 3.5 Internal NAS
15	9AE_LJeJML43	199	1008	Panamax	Details About Panamax Mr4000 8outlets Su

In [ ]:

```
products.to_excel('out.xlsx',
                  sheet_name='Products',
                  startrow=1,
                  startcol=2)
```



	A	B	C	D	E	F	G
1							
2			product_id	price	merchant_id	brand	name
3			AVphzgbJLJeJML43fA0o	104.99	1001	Sanus	Sanus VLF410B1 10-Inch
4			AVpgMuGwLJeJML43KY_c	69	1002	Boytone	Boytone - 2500W 2.1-Ch
5			AVpe9FXeLJeJML43zHrq	23.99	1001	DENAQ	DENAQ - AC Adapter for T
6			AVpfVJXu1cnluZ0-iwTT	290.99	1001	DreamWav	DreamWave - Tremor Po
7			AVphUeKeilAPnD_x3-Be	244.01	1004	Yamaha	NS-SP1800BL 5.1-Chann
8			AVpi6aLv1cnluZ0-Rv8A	254.99	1001	Universal R	Universal Remote Contro
9			AVpf0Nyo1cnluZ0-rzhu	499	1005	Bose	Acoustimass 6 Series V H
10			AWOpOCd2YSSHbkXw07ei	224.99	1001	Samsung	Samsung - 850 PRO 512G
11			AV2Z1Efc-jtxr-f39lm6	139.99	1001	Corsair	Corsair Vengeance LPX 1
12			AVpgNkVc1cnluZ0-yJB6	55.99	1001	Outdoor Te	Outdoor Tech Buckshot I
13			AVpgGPyq1cnluZ0-wbTJ	99.99	1006	Motorola	Motorola Wi-Fi Pet Vide
14			AWACBprXKZqtpbFMVBZo	199.99	1001	Samsung	Details About Samsung G
15			AVpfVI9wilAPnD_xZxH-	89.99	1007	WD	2TB Red 5400 rpm SATA I
16			AVpi9AE_LJeJML43qkYJ	199	1008	Panamax	Details About Panamax M

Saving multiple sheets

If we wanted to write a single `DataFrame` to a single sheet with default formatting then we are done. However, if we want to write multiple sheets and/or multiple `DataFrame` s, then we will need to create an `ExcelWriter` object.

The `ExcelWriter` object is included in the Pandas module and is used to open Excel files and handle write operations. This object behaves almost exactly like the vanilla Python `open` object that we used on previous courses and can be used within a `with` block.

> When the `ExcelWriter` object is executed, any existing file with the same name as the output file will be overwritten.

In [ ]:

```
writer = pd.ExcelWriter('out.xlsx')
```

In [ ]:

```
writer
```

Instead of including the file pathname in the `to_excel` call, we will use the `ExcelWriter` object `writer` instead.

In [ ]:

```
with writer:
    products.to_excel(writer, sheet_name='Products')
```

In [ ]:

```
pd.read_excel('out.xlsx', sheet_name='Products').head()
```

We can now add another `Merchants` sheet simply using the `writer` object:

In [ ]:

```
with writer:
    merchants.to_excel(writer, sheet_name='Merchants')
```

In [ ]:

```
pd.read_excel('out.xlsx', sheet_name='Products').head()
```

In [ ]:

```
pd.read_excel('out.xlsx', sheet_name='Merchants').head()
```

Or we can save multiple sheets at the same time:

In [ ]:

```
with pd.ExcelWriter('out.xlsx') as writer:
    products.to_excel(writer, sheet_name='Products')
    merchants.to_excel(writer, sheet_name='Merchants')
```

In that case the resulting `out.xlxs` file will have two sheets `Products` and `Merchants` .