05_PythonForExcel

May 17, 2022

0.1 Data, Analytics & AI

1 Using Python & Excel Together

1.1 Contents

1.2 Prerequisites

1.2.1 Libraries

We are now familiar with Pandas, Python's counterpart to Excel.

[1]:
```
import pandas as pd
```

The below library is new.

`openpyxl` makes interacting with Excel files from Python simpler.

It is unlikely to be installed by default. By running the below code, you will install it in your current conda environment.

Aside: `!` is a metacommand which tells Jupyter to enter the code following it into the command line

[2]:
```
# Needed if openpyxl not installed
# !conda install openpyxl
```

1.2.2 Data

The data file we are going to be using is called `rates.xlsx` & should be in the `courseware\data` folder.

Rates contains 1 year of exchange rates between GBP and 3 other currencies: EUR, USD, and ROM.

1.3 Reading `.xlsx` files into pandas DataFrames

1.3.1 Overview of read_excel()

Initially, we'll use `pandas` to read Excel files. Then we'll look at manual interaction with `openpyxl` later.

Pandas has a function called `read_excel` which uses `openpyxl` to open an `xlsx` file.

The only argument that we *have* to specify when calling the function is `io`. This is a reference to the file you wish to open, commonly given as a filepath.

We'll be using filepaths, but we could also use an existing Excel file object.

Some of the key optional arguments we can specify include: * sheet_name: allows you to select the sheet(s) from an Excel file you want to load * header: which sheet row contains column headers of the table * names: set column names manually * usecols: select specific columns to load * na_values: specify which data values should be represented as missing (NaN)

The function returns a dictionary of dataframes if multiple sheets are being loaded, or a single dataframe if just one sheet.

Reference: https://pandas.pydata.org/pandas-docs/stable/reference/api/pandas.read_excel.html

1.3.2 Load main sheet

A simple one liner is enough to load in the main sheet of the `rates.xlsx` file.

[3]:
```
df_rates = pd.read_excel('datasets/rates.xlsx')
print(f"data structure that we have read in: {type(df_rates)}")
print(f"df has columns: {df_rates.columns}")
```

```
data structure that we have read in: <class 'pandas.core.frame.DataFrame'>
df has columns: Index(['Date', 'EUR', 'USD', 'RON'], dtype='object')
```

[4]:
```
df_rates
```

[4]:

	Date	EUR	USD	RON
0	2022-05-02	1.1932	1.2557	5.9036
1	2022-05-01	1.1918	1.2561	5.8968
2	2022-04-30	1.1918	1.2561	5.8968
3	2022-04-29	1.1918	1.2561	5.8968
4	2022-04-28	1.1855	1.2430	5.8659
..	…	…	…	…
361	2021-05-06	1.1533	1.3909	5.6816

```
362 2021-05-05  1.1591  1.3915  5.7110
363 2021-05-04  1.1525  1.3854  5.6788
364 2021-05-03  1.1516  1.3870  5.6737
365 2021-05-02  1.1512  1.3909  5.6739

[366 rows x 4 columns]
```

1.3.3 Load all sheets

In order to load all sheets of an Excel file, we simply set `sheet_name=None`. If there are multiple sheets, the result of the `read_excel` function is a dictionary, each key the name of the sheet, each value the Dataframe containing the data of that sheet.

[7]:
```
dict_rates = pd.read_excel('datasets/rates.xlsx', sheet_name = None)
print("data structure that we have read in: ", type(dict_rates))
print("dictionary has keys: ", dict_rates.keys())
```

```
data structure that we have read in:  <class 'dict'>
dictionary has keys:  dict_keys(['excelrates', 'eur', 'usd', 'ron'])
```

We can extract each sheet by giving its name to the dictionary object.

[8]:
```
excelrates = dict_rates["excelrates"]
eur = dict_rates["eur"]
usd = dict_rates["usd"]
ron = dict_rates["ron"]

eur # let's see what have we created
```

[8]:
```
           Date     EUR
0    2022-05-02  1.1932
1    2022-05-01  1.1918
2    2022-04-30  1.1918
3    2022-04-29  1.1918
4    2022-04-28  1.1855
..          …     …
361  2021-05-06  1.1533
362  2021-05-05  1.1591
363  2021-05-04  1.1525
364  2021-05-03  1.1516
365  2021-05-02  1.1512

[366 rows x 2 columns]
```

We can read in each sheet individually by specifying the sheet names. Note the mixed use of a numeric index & the explicit naming of the sheet.

[10]:
```
excelrates = pd.read_excel('datasets/rates.xlsx', sheet_name = 0)
eur = pd.read_excel('datasets/rates.xlsx', sheet_name = 1)
usd = pd.read_excel('datasets/rates.xlsx', sheet_name = 2)
ron = pd.read_excel('datasets/rates.xlsx', sheet_name = "ron")

eur # let's see what have we created
```

[10]:

	Date	EUR
0	2022-05-02	1.1932
1	2022-05-01	1.1918
2	2022-04-30	1.1918
3	2022-04-29	1.1918
4	2022-04-28	1.1855
..	…	…
361	2021-05-06	1.1533
362	2021-05-05	1.1591
363	2021-05-04	1.1525
364	2021-05-03	1.1516
365	2021-05-02	1.1512

[366 rows x 2 columns]

1.4 Writing pandas DataFrames to Excel spreadsheets

Similar to reading, we have a simple command to write back to an excel sheet. It gets trickier when writing multiple sheets & we will have to use iteration to get it done.

1.4.1 Write a single sheet

We'll start by writing a the df_rates dataframe to a single sheet.

[11]:
```
df_rates
```

[11]:

	Date	EUR	USD	RON
0	2022-05-02	1.1932	1.2557	5.9036
1	2022-05-01	1.1918	1.2561	5.8968
2	2022-04-30	1.1918	1.2561	5.8968
3	2022-04-29	1.1918	1.2561	5.8968
4	2022-04-28	1.1855	1.2430	5.8659
..	…	…	…	…
361	2021-05-06	1.1533	1.3909	5.6816
362	2021-05-05	1.1591	1.3915	5.7110
363	2021-05-04	1.1525	1.3854	5.6788
364	2021-05-03	1.1516	1.3870	5.6737
365	2021-05-02	1.1512	1.3909	5.6739

[366 rows x 4 columns]

`to_excel` will create an xlsx file if none exists & write the DataFrame to it. The `.xlsx` part is essential, otherwise pandas gets confused.

[12]:
```
df_rates.to_excel("new_rates.xlsx")
```

Now take a look at your file. It will be in the same place as your workbook. Without sheet names, none will be inferred. We name the sheet using `sheet_name`.

[13]:
```
df_rates.to_excel("new_rates.xlsx",
                  sheet_name="main")
```

1.4.2 Writing multiple sheets

There isn't built in functionality for writing a dictionary to an excel file. We get around this by making a connection to the excel file & writing data to it several times. To do this, we make use of an ExcelWriter object.

To demo this we're using the dict_rates dictionary.

[14]:
```
with pd.ExcelWriter("many_rates.xlsx") as excel_file:
    for sheet, data in dict_rates.items():
        data.to_excel(excel_file, sheet_name=sheet)
```

This logic could easily be abstracted into a function for reuse.

[15]:
```
def dict_to_excel(filename:str, dictonary:dict):
    with pd.ExcelWriter(filename) as excel_file:
        for sheet, data in dictionary.items():
            data.to_excel(excel_file, sheet_name=sheet)
```

1.5 Edit Existing Excel Files

1.5.1 Simple

Read in excel file to DataFrame

[16]:
```
df_rates = pd.read_excel('datasets/rates.xlsx',
                         sheet_name="excelrates")
```

Insert new column, e.g. eur/gbp - ron/gbp differential

[17]:
```
df_rates['eur_rom_diff'] = df_rates['EUR'] - df_rates['RON']
```

We've written to the file new_rates as it avoids corrupting the source sheet. This is just to make teaching easier.

[19]:
```
df_rates.to_excel('datasets/new_rates.xlsx',
                  sheet_name="new_excelrates")
```

1.6 Simple CLI for sheet loading

We have written a small module called `excel_loader.py` which uses user prompts to load sheets from an excel workbook.

To use it, we import the module & run the `get_sheet` function

[23]:
```
import excel_loader

df = excel_loader.get_sheet()
```

```
Enter the name of the file you would like to read
        datasets/new_rates

Enter the sheet you want to load from the file
        new_excelrates
```

[24]:
```
df
```

[24]:

	Unnamed: 0	Date	EUR	USD	RON	eur_rom_diff
0	0	2022-05-02	1.1932	1.2557	5.9036	-4.7104
1	1	2022-05-01	1.1918	1.2561	5.8968	-4.7050
2	2	2022-04-30	1.1918	1.2561	5.8968	-4.7050
3	3	2022-04-29	1.1918	1.2561	5.8968	-4.7050
4	4	2022-04-28	1.1855	1.2430	5.8659	-4.6804
..	…	…	…	…	…	…
361	361	2021-05-06	1.1533	1.3909	5.6816	-4.5283
362	362	2021-05-05	1.1591	1.3915	5.7110	-4.5519
363	363	2021-05-04	1.1525	1.3854	5.6788	-4.5263
364	364	2021-05-03	1.1516	1.3870	5.6737	-4.5221
365	365	2021-05-02	1.1512	1.3909	5.6739	-4.5227

[366 rows x 6 columns]

The module can also be run directly from the command line

[25]:
```
#!python excel_loader.py rates new_excelrates
```

1.7 Python Vs. Excel

If an Excel pro: * Excel for genuinely new, possibly one-off analysis * Python for routine, non-unique tasks on standard sources

If new to both, use whatever you are most comfortable with.

Simple data extraction can be very easily automated.

1.8 Leveraging Python & with Excel

1.8.1 Libraries

[26]:
```
import matplotlib.pyplot as plt
import seaborn as sns
```

1.8.2 Example Task: 30 day moving average exchange rate plot

Pandas DataFranes have lot's of functionalty we can leverage. Say we wanted to geneerate a 30-day moving average of an exchange rate. The `rolling` method makes this easy.

We're going to load in the Euro exchange rate data, calculate a moving average, plot it, and then load that plot back into Excel.

Let's start with `dict_rates`, a sheet we've loaded. We're only interested in the `EUR` table.

[27]:
```
eur = dict_rates['eur']
```

We can directly use the `rolling` method to calculate rolling averages.

Let's construct a 7-day & a 30-day one & compare their appearance.

[28]:
```
seven_day_rolling = eur.rolling(
    on="Date",
    window = 7,
    center=False
).mean()

thirty_day_rolling = eur.rolling(
    on="Date",
    window = 30,
    center=False
).mean()
```

[29]:
```
seven_day_rolling
```

[29]:

	Date	EUR
0	2022-05-02	NaN
1	2022-05-01	NaN
2	2022-04-30	NaN
3	2022-04-29	NaN
4	2022-04-28	NaN
..	…	…
361	2021-05-06	1.156757
362	2021-05-05	1.155843
363	2021-05-04	1.154400
364	2021-05-03	1.153171
365	2021-05-02	1.153071

[366 rows x 2 columns]

1.8.3 Store moving averages in the DataFrame

[30]:
```
eur['SMA_30'] = thirty_day_rolling["EUR"]
eur['SMA_7'] = seven_day_rolling["EUR"]
```

[31]:
```
eur.head(10)
```

[31]:

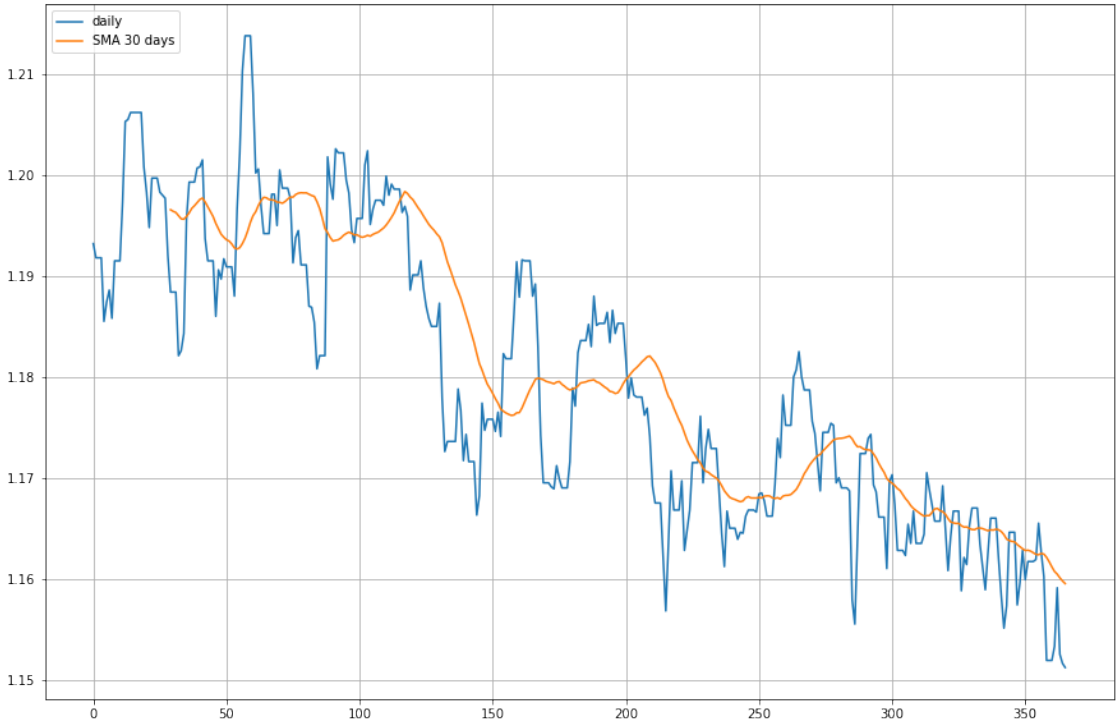
	Date	EUR	SMA_30	SMA_7
0	2022-05-02	1.1932	NaN	NaN
1	2022-05-01	1.1918	NaN	NaN
2	2022-04-30	1.1918	NaN	NaN
3	2022-04-29	1.1918	NaN	NaN
4	2022-04-28	1.1855	NaN	NaN
5	2022-04-27	1.1874	NaN	NaN
6	2022-04-26	1.1886	NaN	1.190014
7	2022-04-25	1.1858	NaN	1.188957
8	2022-04-24	1.1915	NaN	1.188914
9	2022-04-23	1.1915	NaN	1.188871

1.8.4 Visualising the Data

Matplotlib or seaborn allow us to quickly view the data

[32]:
```
import matplotlib.pyplot as plt
plt.figure(figsize=[15,10])
plt.grid(True)
plt.plot(eur['EUR'],label='daily')
plt.plot(eur['SMA_30'],label='SMA 30 days')
plt.legend(loc=2)
```

[32]:
```
<matplotlib.legend.Legend at 0x2b6e2d78700>
```



1.8.5 7 day moving average

[33]:
```
eur['SMA_7'] = eur.iloc[:,1].rolling(window=7).mean()
```

[34]:
```
eur.head(10)
```

[34]:

	Date	EUR	SMA_30	SMA_7
0	2022-05-02	1.1932	NaN	NaN
1	2022-05-01	1.1918	NaN	NaN
2	2022-04-30	1.1918	NaN	NaN
3	2022-04-29	1.1918	NaN	NaN
4	2022-04-28	1.1855	NaN	NaN
5	2022-04-27	1.1874	NaN	NaN
6	2022-04-26	1.1886	NaN	1.190014
7	2022-04-25	1.1858	NaN	1.188957
8	2022-04-24	1.1915	NaN	1.188914
9	2022-04-23	1.1915	NaN	1.188871

[41]:
```
plt.style.use('fivethirtyeight')
plt.figure(figsize=[15,10])
plt.grid(True)
plt.plot(eur['EUR'],label='daily')
plt.plot(eur['SMA_7'],label='SMA 7 days')
```

```
plt.plot(eur['SMA_30'],label='SMA 30 days')
plt.legend(loc=2)

plt.savefig("rolling_averages.png")
```

1.8.6 Write the data

[46]:
```
eur.to_excel('data_and_image.xlsx',
             sheet_name='rate_data')
```

1.8.7 Load the image into Excel

[37]:
```
import openpyxl
```

[49]:
```
wb = openpyxl.load_workbook(filename = 'data_and_image.xlsx')

ws = wb.create_sheet("rate_plot")
```

[50]:
```
img = openpyxl.drawing.image.Image('rolling_averages.png')
img.anchor = 'A1'
ws.add_image(img)
wb.save('data_and_image.xlsx')
```

pd special methods "pandas has many optional dependencies that are only used for specific methods" * ref: https://pandas.pydata.org/docs/getting_started/install.html#optional-dependencies

- pd.read_excel() is dependent on `openpyxl`

1.9 Direct Excel manipulation with OpenPyXl

1.9.1 `openpyxl`

`openpyxl` is a Python library that provides methods to **read** from, **write** to, run **arithmetic operations** in, & **plot graphs** in Excel Files using Python, as well as much more.

[53]:
```
import openpyxl
```

For example, say we wanted to create a new workbook. Within that workbook, we wanted to load in data regarding current GBP, EUR, and RON exchange rates to one sheet, add an image of a rolling average plot in another sheet, and set up an aggregate table in a final sheet.

OpenPyXL lets us do all of that.

First, we create create a workbook object

[51]:
```
wb_obj = openpyxl.Workbook()
```

From the workbook object, we create an active sheet object.

[52]:
```
sheet_obj = wb_obj.active
```

From the sheet object, we can go as far as to select & operate on individual cells.

[53]:
```
cell_obj = sheet_obj.cell(row = 1, column = 1)
```

Before we continue with our example, we'll do a lightning overview of working with workbooks, sheets & cells.

You can think of the OpenPyXl representation of an Excel sheet as similar to a dictionary, but where the index is ordered.

[54]:
```
def print_rows(sheet):
    for row in sheet.iter_rows(values_only=True):
        print(row)
```

[55]:
```
print_rows(sheet_obj)
```

```
(None,)
```

We can act on sheets as we've seen above.

[56]:
```
sheet_obj["A1"] = "Data"
```

[57]:
```
print_rows(sheet_obj)
```

```
('Data',)
```

[58]:
```
sheet_obj["B1"] = 100
```

[59]:
```
print_rows(sheet_obj)
```

```
('Data', 100)
```

[60]:
```
wb_obj.save("createsrates.xlsx")
```

further reading

- https://blog.quantinsti.com/python-trading/
- https://www.codementor.io/@ivyli/beginner-python-financial-project-see-behind-the-fx-rate-122874sico