

Alternative and Quantitative Investment Strategies

Grenoble INP - Ensimag

Agenda for today

- Introduction to trades data
- Pandas basics
- Data handling using Pandas
- Practice: Prepare the dataset and use Pandas to explore and handle a dataset of crypto trade prices

Objectives:

1. Be familiar with trades data and time series
2. Be comfortable with the main functions of the Pandas Python module
3. Be able to manipulate a dataset and answer functional questions using it

Part 1

Introduction to trades data

Definitions

Trade: “A financial transaction consisting in buying or selling a financial instrument to a counterparty.”

A trade's usual characteristics are:

- The financial instrument traded, also called the *underlying instrument*
- The price of the transaction
- The number of instruments traded (purchased or sold), also called the *amount*
- The timestamp of the transaction
- The trade's side, i.e. if it's a purchase or a sale.

For practical identification and storage purposes, each trade is often assigned a unique identifier (numeric or not).

The dataset

The dataset on which you will work in this session is a list of trades, really executed on several exchanges on October 24 between 2:00 and 4:00 pm.

The financial instruments are cryptocurrency pairs.

E.g.: BTC-USD

Each exchange is represented by a 4-letter code.

The dataset

Example: If an investor buys 10 BTC-USD at the price of 20,000:

- The *underlying* instrument is the pair BTC-USD
- BTC is the *base* asset
- USD is the *quote* asset
- The trade *price* is 20,000
- The trade *amount* is 10. The amount is expressed in the base asset.
- The trade *volume* is 200,000. The volume is expressed in the quote asset and is equal to the price multiplied by the amount.
- The trade is a purchase (not a sale)

Part 2

Pandas basics

Pandas

Pandas is a Python library created in 2008 and is used for working with data sets.

This module is particularly useful for **analyzing, cleaning, exploring**, and **manipulating** data.

```
import pandas as pd
```



The name Pandas has several origins and is both a contraction of “Panel Data” and “Python Data Analysis”.

Pandas' types - Series

Pandas' first main type is a ***Series***.

A Pandas Series is a one-dimensional array holding data of any type.

If nothing else is specified, the values are accessible like in any standard (or Numpy) array, and the first value's index is 0.

```
values = [12, 27, 33]

my_series = pd.Series(values)

snd_element = my_series[1] # -> 27
```

0	12
1	27
2	33

Indexes in Pandas are also called *Labels*.

Pandas' types - Series

You can also customize the Series by specifying your own labels and giving the value a name.

```
values = [12, 27, 33]

my_series = pd.Series(data = values,
                      index = ['A', 'B', 'C'],
                      name = "MySerie")
```

	<i>MySerie</i>
A	12
B	27
C	33

Supported value types or labels are float, int, bool, datetime64[ns], timedelta[ns], and object.

Pandas' types - DataFrame

In Pandas, multi-dimensional tables are called **DataFrames**.

A DataFrame is composed of several Series sharing the same labeling.

One of the ways to create a DataFrame is through a dictionary

```
data = {"Values": [12, 27, 33], "Address": ["0x123", "0x584", "0x695"]}
df = pd.DataFrame(data = data, index = ['A', 'B', 'C'])
```

	Values	Address
A	12	0x123
B	27	0x584
C	33	0x695

DataFrame - Accessing the data

1. Access to one or several columns:

```
df.Values  
df["Values"]  
df[["Values", "Address"]]
```

2. Access to one or several rows:

```
df.loc["A"]  
df.loc[["A", "C"]]
```

3. Access to a single value:

```
df.loc["A", "Values"]
```

DataFrame - Accessing the data

4. Access to values by their index:

```
df.iloc[0] # 1st row  
df.iloc[-1] # last row  
df.iloc[0, 2] # 1st row, 3rd column
```

5. Access to a range (or slice) of values:

```
df.iloc[1:5] # rows B,C,D E  
df.iloc[:, 1] # column Address  
df.iloc[5:, [0,2]]
```

	BlockNb	Address	AverageSize
A	12	0x123	2.5
B	27	0x584	6.4
C	33	0x695	1.2
D	58	0x675	3.4
E	98	0x254	2.5
F	45	6x695	3.1
G	25	0x584	1.1
H	48	0x978	0.5

DataFrame - Accessing the data

6. Conditional access to values:

```
df[df.Address == "0x584"] # rows B and G
df[df.Address.isin(["0x584", "0x695"])] # rows B, C and G
df[df.AverageSize <= 2] # rows C, G and H

df[~df.Address.isin(["0x584", "0x695"])&(df.AverageSize <= 2)]
# others methods: any, all, between
```

	BlockNb	Address	AverageSize
A	12	0x123	2.5
B	27	0x584	6.4
C	33	0x695	1.2
D	58	0x675	3.4
E	98	0x254	2.5
F	45	6x695	3.1
G	25	0x584	1.1
H	48	0x978	0.5

7. Modifications using loc:

```
df.loc['A', "Address"] = "0x555"
df.loc[df.AverageSize <= 2, "BlockNb"] = 0
```

Step 0 – Reading the data

Usually, we don't create a DataFrame from scratch but we get the data from an external source.

The Pandas module has a large number of reading methods. The most used ones are:

- `read_csv`, `read_excel`
- `read_json`
- `read_html`
- `read_clipboard`
- `read_table`
- `read_sql`, `read_sql_query`, `read_sql_table`

These methods each have many useful parameters: `parse_dates`, `index_col`, `encoding`, `float_precision`, ...

Step 1 –Analyzing the data

Once you read your data set, the first thing is to inspect it and take a first look at the data.

In Pandas, you have several methods to help you do this:

- `info`, `describe`: give general information about the dataset and its columns
- `head`, `tail`: focus on the first (resp. last) rows of the dataset
- `set_index`, `reset_index`: change the labeling
- other helpful methods: `unique`, `nunique`, `duplicated`, `value_counts`, `sort_values`, `sort_index`, `nsmallest`, `nlargest`, `idxmin`, `idxmax`, ...

Step 2 – Cleaning the data

Data cleaning means fixing problematic data before starting to use them.

Problematic data could be:

- Empty cells or NaNs
- Wrong format
- Duplicates
- Wrong data, outliers

If the dataset is not clean, it can lead to **errors** or **misinterpretations** while you explore and manipulate the data in the next steps.

Helpful methods: `astype`, `fillna`, `dropna`, `drop_duplicates`, ...

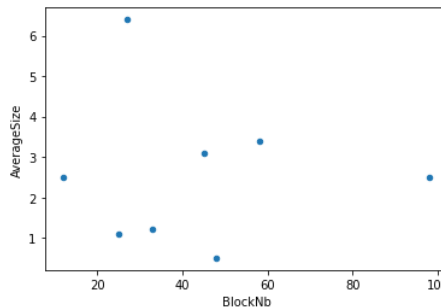
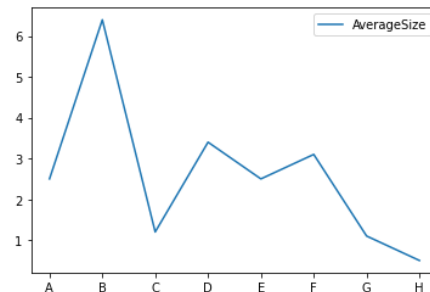
Step 3 – Exploring the data

Pandas is linked to Numpy and most of the Numpy methods can be directly used on the DataFrame.

Examples: abs, mean, median, std, corr

It's also linked to Matplotlib.

```
df.AverageSize.plot()  
df.plot.scatter(x = "BlockNb", y = "AverageSize")  
# df.AverageSize.plot.hist() ...
```



Part 3

Use Pandas to explore and handle
a large dataset

Step 4.1 – Enrich the data set

Finally, we are ready to start handling the data to answer our questions.

1. Creating new columns

```
df["NextBlock"] = df.BlockNb + 1
```


```
def add_one(x):  
    return x + 1
```

```
df["NextBlock"] = df.BlockNb.apply(add_one)
```

or

```
df["NextBlock"] = df.BlockNb.apply(lambda x: x+1)
```

	BlockNb	Address	AverageSize	NextBlock
A	12	0x123	2.5	13
B	27	0x456	6.4	28
C	33	0x789	1.2	34
D	58	0x123	3.4	59
E	98	0x456	1.1	99



Step 4.1 – Enrich the data set

1. Creating new columns

```
df["SizeRange"] = pd.cut(df.AverageSize, np.arange(0, 10, 0.1))  
df['Weight'] = pd.qcut(df.AverageSize, 3, labels=['Light', 'Medium', 'Heavy'])  
df["EasyToCopy"] = df.Weight.map({"Light":True, "Medium":True, "Heavy":False})
```

	BlockNb	Address	AverageSize		SizeRange	Weight	EasyToCopy
A	12	0x123	2.5		(2.4, 2.5]	Medium	True
B	27	0x456	6.4		(6.3, 6.4]	Heavy	False
C	33	0x789	1.2	←	(1.1, 1.2]	Light	True
D	58	0x123	3.4		(3.3, 3.4]	Heavy	False
E	98	0x456	1.1		(1.0, 1.1]	Light	True

Step 4.1 – Enrich the data set

NB: Operations on string or datetime columns

	BlockNb	Address	AverageSize
A	12	0x123	2.5
B	27	0x584	6.4
C	33	0x695	1.2
D	58	0x675	3.4
E	98	0x254	2.5

	Price	Date
0	12.5	2023-01-01 08:00:00
1	13.1	2023-01-02 10:00:00
2	12.7	2023-01-03 14:00:00
3	11.9	2023-01-04 18:00:00

```
df.Address.str[2:]  
# df.Address.str[2:].astype(int)  
df.Address.str.upper()
```

```
df.Date.dt.hour # extract the hours from the dates  
  
df.Date.apply(lambda d: d + dt.timedelta(hours=12))  
# add 12 hours to each date
```

Step 4.2 – Handling the data

2. Change the indexing – single indexing

```
df.reset_index(inplace = True)  
df.set_index("BlockNb")
```


BlockNb Address AverageSize								BlockName Address AverageSize				
BlockName				BlockName BlockNb Address AverageSize				BlockNb				
A	12	0x123	2.5	0	A	12	0x123	2.5	12	A	0x123	2.5
B	27	0x456	6.4	1	B	27	0x456	6.4	27	B	0x456	6.4
C	33	0x789	1.2	2	C	33	0x789	1.2	33	C	0x789	1.2
D	58	0x123	3.4	3	D	58	0x123	3.4	58	D	0x123	3.4
E	98	0x456	1.1	4	E	98	0x456	1.1	98	E	0x456	1.1

Step 4.2 – Handling the data

2. Change the indexing – multiple-indexing

```
df.set_index(["Address", "BlockNb"])
```

BlockNb		Address	AverageSize
BlockName			
A	12	0x123	2.5
B	27	0x456	6.4
C	33	0x789	1.2
D	58	0x123	3.4
E	98	0x456	1.1



BlockName		AverageSize
Address	BlockNb	
0x123	12	A 2.5
	58	D 3.4
0x456	27	B 6.4
	98	E 1.1
0x789	33	C 1.2

Step 4.2 – Handling the data

2. Change the indexing – stack/unstack

```
df.set_index(["Address", "BlockNb"], inplace = True)  
df.unstack(level=1)
```

				BlockName				
Address	BlockNb	BlockName		Address	BlockNb			
0	0x123	12	A	→	0x123	12	A	
1	0x456	27	B			58	D	
2	0x789	33	C			0x456	27	B
3	0x123	58	D			98	E	
4	0x456	98	E			0x789	33	C


		BlockName					
		BlockNb	12	27	33	58	98
		Address					
	0x123	A	NaN	NaN	D	NaN	
	0x456	NaN	B	NaN	NaN	E	
	0x789	NaN	NaN	C	NaN	NaN	

Step 4.2 – Handling the data


3. Grouping and aggregating the data

It's always the same pattern:

```
df.grouping_method(...).aggregating_method(...)
```



This method defines
according to which criteria
you want to group the data.



This method defines the
aggregation method to use
for each group.

Step 4 – Handling the data

3.1 Aggregations by a single discrete criterion

Total daily volume by the exchange type

```
df.groupby('IsCentralized').TotalDailyVolume.sum()
```

Out

TotalDailyVolume	
IsCentralized	
False	28508
True	5351785

	Exchange	Rank	TotalDailyVolume	IsCentralized
0	cbse	1	4687512	True
1	stmp	2	45687	True
2	huob	3	13694	True
3	gmni	4	468976	True
4	usp3	5	11514	False
5	sush	6	15648	False
6	krkn	7	126548	True
7	btrx	8	4687	True
8	plnx	9	1346	False
9	ftxx	10	4681	True

Step 4 – Handling the data

3.2 Aggregations by a single non-discrete criterion

Average **and** total daily volume by the range of ranking

```
rank_range = pd.cut(df.Rank, bins = [0,3,5,7,10])  
df.groupby(rank_range).TotalDailyVolume.agg(['sum', 'mean'])
```

Out

	Total Daily Volume	Average Daily Volume
Rank		
(0, 3]	4746893	1.582298e+06
(3, 5]	480490	2.402450e+05
(5, 7]	142196	7.109800e+04
(7, 10]	10714	3.571333e+03

	Exchange	Rank	TotalDailyVolume	IsCentralized
0	cbse	1	4687512	True
1	stmp	2	45687	True
2	huob	3	13694	True
3	gmni	4	468976	True
4	usp3	5	11514	False
5	sush	6	15648	False
6	krkn	7	126548	True
7	btrx	8	4687	True
8	plnx	9	1346	False
9	ftxx	10	4681	True

Step 4 – Handling the data

3.3 Aggregations by multiple criteria

Total daily volume by type of exchange and range of ranking

```
rank_range = pd.cut(df.Rank, bins = [0,5,10])  
df.groupby(['IsCentralized',rank_range]).TotalDailyVolume.sum()
```

Out

TotalDailyVolume		
IsCentralized	Rank	
False	(0, 5]	11514
	(5, 10]	16994
True	(0, 5]	5215869
	(5, 10]	135916

	Exchange	Rank	TotalDailyVolume	IsCentralized
0	cbse	1	4687512	True
1	stmp	2	45687	True
2	huob	3	13694	True
3	gmni	4	468976	True
4	usp3	5	11514	False
5	sush	6	15648	False
6	krkn	7	126548	True
7	btrx	8	4687	True
8	plnx	9	1346	False
9	ftxx	10	4681	True

Step 4 – Handling the data

4. Grouping and aggregating data over time

4.1 Using groupby

```
df.groupby(pd.Grouper(key="timestamp", freq = "1D")).price.mean()
```

Out

price	
timestamp	
2022-05-01	38063.870109
2022-05-02	38672.806200
2022-05-03	38242.586549
2022-05-04	38813.334858
2022-05-05	38413.086732
2022-05-06	36139.395325

You can choose any frequency
you want, e.g. « 1W », « 15Min »,
« 1B », « 1Y » ...

	timestamp	price
0	2022-05-01 00:00:00	37644.135964
1	2022-05-01 00:01:00	37642.195815
2	2022-05-01 00:02:00	37671.034620
3	2022-05-01 00:03:00	37690.006935
4	2022-05-01 00:04:00	37728.377096
...
86395	2022-06-29 23:55:00	20076.720445
86396	2022-06-29 23:56:00	20087.645988
86397	2022-06-29 23:57:00	20080.135249
86398	2022-06-29 23:58:00	20091.394314
86399	2022-06-29 23:59:00	20093.026235

Step 4 – Handling the data

4. Grouping and aggregating data over time

4.2 Using resample

```
df.resample(rule="1D", on="timestamp").price.mean()
```

Out

price	
timestamp	
2022-05-01	38063.870109
2022-05-02	38672.806200
2022-05-03	38242.586549
2022-05-04	38813.334858
2022-05-05	38413.086732
2022-05-06	36139.395325

You can choose any frequency you want, e.g. « 1W », « 15Min », « 1B », « 1Y » ...

	timestamp	price
0	2022-05-01 00:00:00	37644.135964
1	2022-05-01 00:01:00	37642.195815
2	2022-05-01 00:02:00	37671.034620
3	2022-05-01 00:03:00	37690.006935
4	2022-05-01 00:04:00	37728.377096
...
86395	2022-06-29 23:55:00	20076.720445
86396	2022-06-29 23:56:00	20087.645988
86397	2022-06-29 23:57:00	20080.135249
86398	2022-06-29 23:58:00	20091.394314
86399	2022-06-29 23:59:00	20093.026235

Step 4 – Handling the data

5. Pivot tables

```
df.pivot_table(index="pair", columns="exchange", values="price")
```

Out

exchange	cbse	gmni	krkn	stmp
pair				
btc-usd	36139.395325	35848.322476	32653.683893	34537.983572
eth-usd	1548.445340	1549.452750	1547.575820	1553.552770

	pair	exchange	price
0	btc-usd	cbse	36139.395325
1	btc-usd	gmni	35848.322476
2	btc-usd	stmp	34537.983572
3	btc-usd	krkn	32653.683893
4	eth-usd	cbse	1548.445340
5	eth-usd	gmni	1549.452750
6	eth-usd	stmp	1553.552770
7	eth-usd	krkn	1547.575820

NB: when needed, you can enter a parameter `aggfunc` to specify a value aggregation method.

Part 4 - Practice

Prepare the data and use Pandas to explore and handle a large dataset

Practice

Complete the jupyter notebooks named **TP1_Preparing_the_data_solutions.ipynb** and **TP2_Handling_trades_data.ipynb**.

The data set can be found in the following CSV file: **TP1_trades_data.csv**.



It's forbidden to use a loop (`for`, `while`) in both notebooks.