How to Easily Fetch Binance Historical Trades Using Python

**Create a reliable script to extract historical trade data from Binance**

[[](https://thiago-gcandido.medium.com/?source=post_page-----174a6569cebd--------------------------------)](https://thiago-gcandido.medium.com/?source=post_page-----174a6569cebd--------------------------------)

[Thiago Candido](https://thiago-gcandido.medium.com/?source=post_page-----174a6569cebd--------------------------------)

Follow

[May 6, 2020](https://betterprogramming.pub/how-to-easily-fetch-your-binance-historical-trades-using-python-174a6569cebd?source=post_page-----174a6569cebd--------------------------------) · 6 min read



Photo by [Austin Distel](https://unsplash.com/@austindistel?utm_source=unsplash&utm_medium=referral&utm_content=creditCopyText) on [Unsplash](https://unsplash.com/s/photos/trading?utm_source=unsplash&utm_medium=referral&utm_content=creditCopyText)

*Note: this article is provided for entertainment and educational purposes only and is not intended as financial advice.*

As an ex-CTO of a cryptocurrency exchange, I had my fair share of messing around with several major exchanges’ APIs. In this article, I will guide you through the process of creating a reliable Python script to extract historical trade data from Binance.

Rationale

When backtesting a trading strategy, that is, for executing our strategy with past data and analyzing the returns and other important factors, we have to make sure that we have the appropriate kind of data to work with. The process isn’t always so simple, given that some strategies need level book data, others may do just fine with one-hour candles, and elements like infrastructure, availability, and connectivity can be a lot different depending on the type of data that you need.

There’s no need to worry if you don’t have the slightest clue about what these *level book data o*r *one-hour candles*means, as we will not be touching on those in this article.

The data needed is mainly defined by the frequency of the trading strategy. Trading strategy categories are a topic for a whole other subject that I will be writing about in my Algotrading series, but you can find some reliable information in [Investopedia.](https://www.investopedia.com/articles/active-trading/11/four-types-of-active-traders.asp)

Ok, but why is this article only about fetching “trade” data, and why are we using Binance API? You, the reader, must be a little bit confused about my article’s content.

Data frequency and Binance

Trade data endpoints are mainly available in, I would say, 99.99% of the exchanges. It’s granular, providing enough detail (in somevery particular cases) for backtesting [high-frequency trading (HFT)](https://www.investopedia.com/terms/h/high-frequency-trading.asp)strategies, as well to serve as the building blocks of [OHLC candles](https://www.investopedia.com/terms/o/ohlcchart.asp) (1S to 24H, or more, if that’s what you wish).

Trade data is versatile and allows for a lot of experimenting with strategies of different frequencies.

Why [Binance](https://binance.com/)? That’s just because it’s one of the exchanges I tend to backtest due to its volume. In no way am I affiliated with Binance. We’ll check out other exchanges in my Algotrading series.

“Talk is cheap. Show me the code.” — Linus Torvalds

What We’ll Be Coding

We are going to create a Python script that receives as command-line arguments the pair symbol, a starting date, and an ending date. It will output to disk a CSV file containing all the trades. The process can be detailed by the following steps:

* Parse symbol, starting\_date, and ending\_datearguments.
* Fetch the first trade that happened on the starting date, to get the first trade\_id.
* Loop fetching 1,000 trades (Binance API limit) per request until ending\_date is reached.
* Finally, save the data to disk. We will save it in CSV for the example, but you have several other options to choose from.

We will be using the pandas, requests , time, sys, and datetime libraries. In the code snippets, no error verification will be shown because it doesn’t add any value to the explication. Of course, you can check out the full code on [GitHub](https://github.com/tgcandido/binance-data-fetcher).

Coding Time

Parsing the arguments

The script will use the following arguments:

* symbol: The symbol of the trading pair, defined by Binance. It can be [queried here](https://api.binance.com/api/v1/ticker/allPrices), or it may be copied from the URL of the Binance web app, excluding the \_ character.

https://miro.medium.com/max/60/1*_pB8bKGbi5_ofHVFOouj8A.png?q=20

https://miro.medium.com/max/306/1*_pB8bKGbi5_ofHVFOouj8A.png

Remove the ‘\_’ from the last part of the URL and you get the symbol

* starting\_date and ending\_date: Self-explanatory. The expected format is mm/dd/yyyy , or, in Python slang, %m/%d/%Y.

To get the arguments, we’ll use the built-in sys (nothing too fancy around here), and to parse the date, we will be using the datetime library.

symbol = sys.argv[1]  
starting\_date = datetime.strptime(sys.argv[2], '%m/%d/%Y')  
ending\_date = datetime.strptime(sys.argv[3], '%m/%d/%Y') + timedelta(days=1) - timedelta(microseconds=1)

We are adding one day and subtracting one microsecond so that the ending\_date time portion is always at 23:59:59.999, making it more practical to get same-day intervals.

Fetching trades

With [Binance’s API](https://github.com/binance-exchange/binance-official-api-docs/blob/master/rest-api.md) and using the [aggTrades](https://github.com/binance-exchange/binance-official-api-docs/blob/master/rest-api.md#compressedaggregate-trades-list) endpoint, we can get at most 1,000 trades in one request, and if we use start and end parameters, they can be at most one hour apart. After some failures, by fetching using time intervals (at some point or another, the liquidity would go crazy and I would lose some precious trades), I decided to try the from\_id strategy.

The aggTrades endpoint is chosen because it returns the compressed trades. In that way, we won’t lose any precious information.

Get compressed, aggregate trades. Trades that fill at the same time, from the same order, with the same price will have the quantity aggregated.

The from\_id strategy goes like this: We are going to get the first trade of the starting\_dateby sending date intervals to the endpoint. After that, we will fetch 1,000 trades, starting with the first fetched trade ID. Then, we will check if the last trade happened after our ending\_date. If so, we have gone through all the time period and we can save the results to file. Otherwise, we will update our from\_id variable to get the last trade ID and start the loop all over again.

Ugh, enough talking, let’s code.

Fetching the first trade ID

First, we create a new\_end\_date . That’s because we are using the aggTrades by passing a startTimeand an endTimeparameter. For now, we only need to know the first trade ID of the period, so we are adding 60 seconds to the period. In low liquidity pairs, this parameter can be changed because there is no guarantee that a trade occurred in the first minute of the day that’s been requested.

Then, parse the date using our helper function to convert it to a Unix millisecond representation by using the calendar.timegm function. The timegm function is preferred because it keeps the date in UTC.

def get\_unix\_ms\_from\_date(date):  
 return int(calendar.timegm(date.timetuple()) \* 1000 + date.microsecond/1000)

The request’s response is a list of trade objects sorted by date, with the following format:

So, as we need the first trade ID**,**we will be returning the response[0]["a"] value.

Main loop

Now that we have the first trade ID, we can fetch trades 1,000 at a time, until we reach our ending\_date. The following code will be called inside our main loop. It will perform our request using the from\_id parameter, ditching the startDate and endDate parameters.

And now, here’s our main loop, which will perform the requests and create our [DataFrame](https://pandas.pydata.org/pandas-docs/stable/reference/api/pandas.DataFrame.html).

We check if the current\_time that contains the date of the latest trade fetched is greater than our to\_date , and if so, we:

* fetch the trades using the from\_id parameter
* update the from\_id and current\_time parameters, both with information from the latest trade fetched
* print a nice debug message
* [pd.concat](https://pandas.pydata.org/pandas-docs/stable/reference/api/pandas.concat.html) the trades fetched with the previous trades in our DataFrame
* and [sleep](https://docs.python.org/3/library/time.html) a little so that Binance won’t give us an ugly 429 HTTP response

Cleaning and saving

After assembling our DataFrame, we need to perform a simple data cleaning. We will remove the duplicates and trim the trades that happened after our to\_date (we have that problem because we’re fetching in chunks of 1,000 trades, so it’s expected that we get some trades executed after our target end date).

We can encapsulate our trim functionality:

def trim(df, to\_date):  
 return df[df['T'] <= get\_unix\_ms\_from\_date(to\_date)]

And perform our data cleaning:

df.drop\_duplicates(subset='a', inplace=True)  
df = trim(df, to\_date)

Now, we can save it to file using the [to\_csv](https://pandas.pydata.org/pandas-docs/stable/reference/api/pandas.DataFrame.to_csv.html) method:

filename = f'binance\_\_{symbol}\_\_trades\_\_from\_\_{sys.argv[2].replace("/", "\_")}\_\_to\_\_{sys.argv[3].replace("/", "\_")}.csv'  
df.to\_csv(filename)

We can also use other data storage mechanisms, such as Arctic.

Bonus: Verifying Your Data

It’s important that we can trust our data when working with trading strategies. We can easily do that with the fetched trade data by applying the following for verification:

In the snippet, we convert our DataFrame to a [NumPy array](https://numpy.org/doc/stable/reference/arrays.html) and iterate row by row, checking if the trade ID is incremented by 1 each row.

Binance trade IDs are numbered incrementally and are created for each symbol, so it’s really easy to verify if your data is correct.

Conclusion

The first step to create a successful trading strategy is to have the right data.

My Algotrading series is a work in progress, so I welcome any feedback or suggestion you leave me in the comment section. You can check out the full code of this tiny tutorial in my GitHub [repository](https://github.com/tgcandido/binance-data-fetcher).

I hope you enjoyed reading this post. Thank you for your time.

Take care and keep coding!