yayFinPy

Yet Another Yahoo! Finance Python API: Based on yfinance Python API

Team 13: Final Project Report on fixing broken API Design and (Re) Implementation

17-780

API DESIGN AND IMPLEMENTATION

TEAM MEMBERS

1. Vikramraj Sitpal (vsitpal)
2. Chirag Sachdeva (csachdev)
3. Tianyang Zhan (tzhan)
4. Shubham Gupta (shubhamg)
5. Vasudev Luthra (vasudevl)

PROGRAMMING LANGUAGE

python powered

```
print("Hello, world!")
```

PROJECT GOAL

Find and fix a publicly available and used API which is broken in terms of design principles taught by Josh in the course. As a result, design a replacement API that fixes the flaws, and prototype and test the replacement API.

API SELECTED

yfinance
Documentation: https://pypi.org/project/yfinance/
New name for our API: **yayFinPy**

INTRODUCTION

yfinance[1] is a python package that sees 99,000 monthly and is used by various stock market related entities to obtain information about stocks, historic data, which they use for various analysis and purposes. We found a significant number of flaws and challenges in the API which makes it a suitable API to target for improvement. We believe that improving this API would have a significant impact on the users of this API. In addition to fixing the API, we completely restructured the API to make it approachable and easy to use and added several additional features like stock sentiment analysis, tweets, custom portfolio management et cetera.

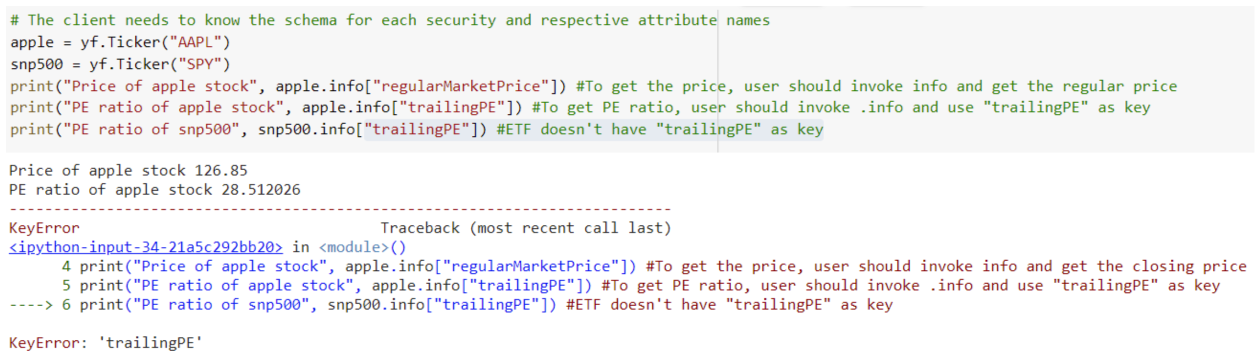
In finance, a security[2] is a financial instrument, typically any financial asset that can be traded. These can be Stocks, ETFs, Mutual Funds, Currency, Cryptocurrency, Commodities etc. A ticker symbol is an abbreviation used to uniquely identify publicly traded/available shares of a particular type of security. Examples: ORCL (Stock), DOGE-USD (cryptocurrency), SPY (ETF), et cetera.

FLAWS IN THE YFINANCE API

We identified that yfinance was a widely used public API in the domain of finance. On analysing a lot of user codes, we realized that most users were using this API for either downloading or pulling the historical price or volume data for specific securities (mostly limited to Stocks, Currency and ETFs). In addition to this, a lot of users were using this data either to do technical analysis or deploy machine learning algorithms.

Based on the issues encountered by users, we identified key areas where the API had some serious issues. These were classified into 3 main categories:

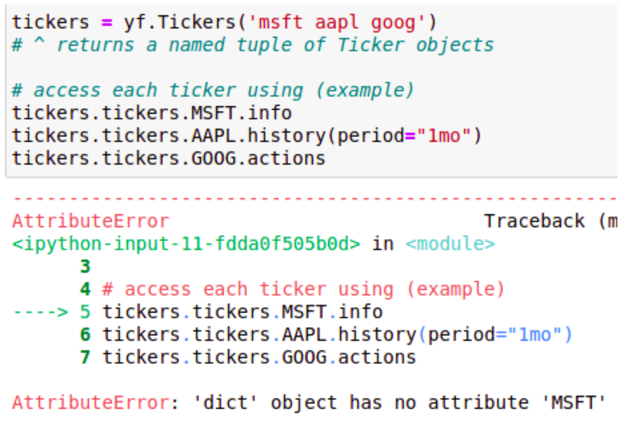
Structure : The original finance API lacks the key layer of abstraction and groups all securities as tickers.
The yFinance API had 2 classes Ticker and Tickers for a single security and for a group of securities respectively. This implied that the user must be familiar with each of the securities and their respective attributes. Some method calls on different tickers sometimes gave errors as shown below.



```
# The client needs to know the schema for each security and respective attribute names
apple = yf.Ticker("AAPL")
snp500 = yf.Ticker("SPY")
print("Price of apple stock", apple.info["regularMarketPrice"]) #To get the price, user should invoke info and get the regular price
print("PE ratio of apple stock", apple.info["trailingPE"]) #To get PE ratio, user should invoke .info and use "trailingPE" as key
print("PE ratio of snp500", snp500.info["trailingPE"]) #ETF doesn't have "trailingPE" as key
```

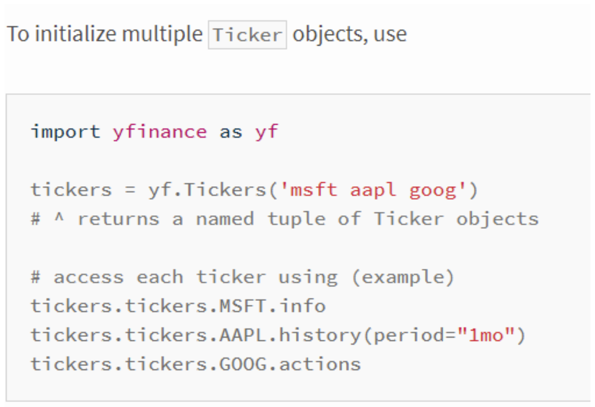
```
Price of apple stock 126.85
PE ratio of apple stock 28.512026
---------------------------------------------------------------------------
KeyError                                  Traceback (most recent call last)
<ipython-input-34-21a5c292bb20> in <module>()
      4 print("Price of apple stock", apple.info["regularMarketPrice"]) #To get the price, user should invoke info and get the closing price
      5 print("PE ratio of apple stock", apple.info["trailingPE"]) #To get PE ratio, user should invoke .info and use "trailingPE" as key
----> 6 print("PE ratio of snp500", snp500.info["trailingPE"]) #ETF doesn't have "trailingPE" as key

KeyError: 'trailingPE'
```

Even though the API provides a way to work with multiple tickers at the same time, that doesn't even work as stated in documentation.



```
tickers = yf.Tickers('msft aapl goog')
# ^ returns a named tuple of Ticker objects

# access each ticker using (example)
tickers.tickers.MSFT.info
tickers.tickers.AAPL.history(period="1mo")
tickers.tickers.GOOG.actions
```

```
---------------------------------------------------------------------------
AttributeError                            Traceback (m
<ipython-input-11-fdda0f505b0d> in <module>
      3
      4 # access each ticker using (example)
----> 5 tickers.tickers.MSFT.info
      6 tickers.tickers.AAPL.history(period="1mo")
      7 tickers.tickers.GOOG.actions

AttributeError: 'dict' object has no attribute 'MSFT'
```



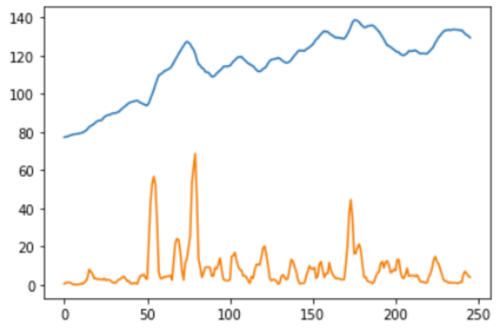
To initialize multiple `Ticker` objects, use

```
import yfinance as yf

tickers = yf.Tickers('msft aapl goog')
# ^ returns a named tuple of Ticker objects

# access each ticker using (example)
tickers.tickers.MSFT.info
tickers.tickers.AAPL.history(period="1mo")
tickers.tickers.GOOG.actions
```

Utility : The API in its present form provides very limited functionality. Thus, the client has to implement several common functionalities after calling this API. It should make it easy for the users to perform common things like technical analysis, manage custom portfolios, get sentiment from public forums, tweets etc.

After analysing different client codes (an example can be seen below) for this API, we believe we can make it easier for users to perform common operations and improve the API by providing additional utility functions. For example, the weekly moving average of the stock price is a common metric that a lot of client codes seem to have and currently the yfinance API doesn't provide any method to perform such operations. There are other similar additional methods that this API should be providing to make it easier for the users to use this API.

```python
import statistics
data = apple.history(period="1y")
historical_price = data["Close"]
variance = []
moving_averages = []
window_size = 7
i = 0
while i < len(historical_price) - window_size + 1:
    this_window = historical_price[i : i + window_size]
    moving_averages.append(statistics.mean(this_window))
    variance.append(statistics.variance(this_window))
    i += 1
plt.plot(moving_averages)
plt.plot(variance)
```



Handling exception and edge cases : It fails to handle several edge cases, and considers any ticker as a valid ticker. Even valid tickers sometimes throw exceptions (recent IPO), fail silently for some method calls or return wrong data as shown in example code snippets below.

```python
#gives some random data
apple.history(period="")
yf.Ticker("SPY").institutional_holders
```

	0	1
0	Net Assets	363.96B
1	NAV	422.23
2	PE Ratio (TTM)	NaN
3	Yield	1.33%
4	YTD Daily Total Return	13.21%

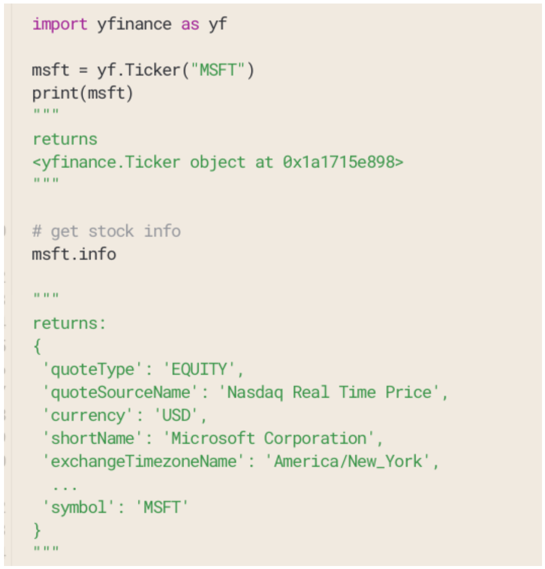
```python
#Different securities
yf.Ticker("AAPL").info["forwardPE"]
yf.Ticker("DOGE-USD").info["forwardPE"] #No error
```

```python
#Invalid ticker (Fails silently)
yf.Ticker("DOGCoin")
```

```
yfinance.Ticker object <DOGCOIN>
```

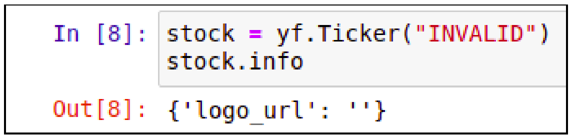
Some instances and principles to demonstrate why this API is broken:

1. The `Ticker()` (used in case of single ticker parameter) or `Tickers()` (used in case of a list of ticker parameters) classes can be made richer in the sense that a lot of features can be removed/changed in order to make it work seamlessly. Right now the functionality between these two classes differs and the way we interact with each of them is inconsistent. These classes can be made more consistent in their interfaces and functionality.
   Principles of concern: U2, U5



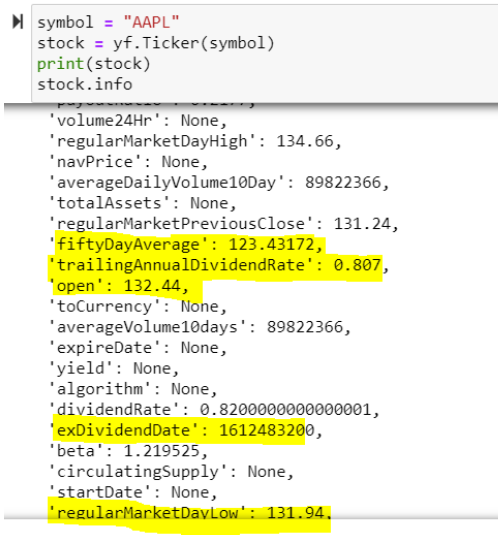
```
import yfinance as yf

msft = yf.Ticker("MSFT")
print(msft)
"""
returns
<yfinance.Ticker object at 0x1a1715e898>
"""

# get stock info
msft.info

"""
returns:
{
 'quoteType': 'EQUITY',
 'quoteSourceName': 'Nasdaq Real Time Price',
 'currency': 'USD',
 'shortName': 'Microsoft Corporation',
 'exchangeTimezoneName': 'America/New_York',
  ...
 'symbol': 'MSFT'
}
"""
```

2. The input is required to exactly match the ticker symbols, and fails silently for invalid ticker symbols or for new IPOs. Does not show any errors for invalid Ticker initialization as shown in the screenshot.
Principles of concern: Q4, Q5, E1



```
In [8]: stock = yf.Ticker("INVALID")
        stock.info

Out[8]: {'logo_url': ''}
```

3. No __str__() method is provided for the objects. The .info method returns information in an unstructured format with inconsistent types. For example, for Apple, the format of date, the type of volume (int) or price (float) are all inconsistent..
Principles of concern: M1, M7

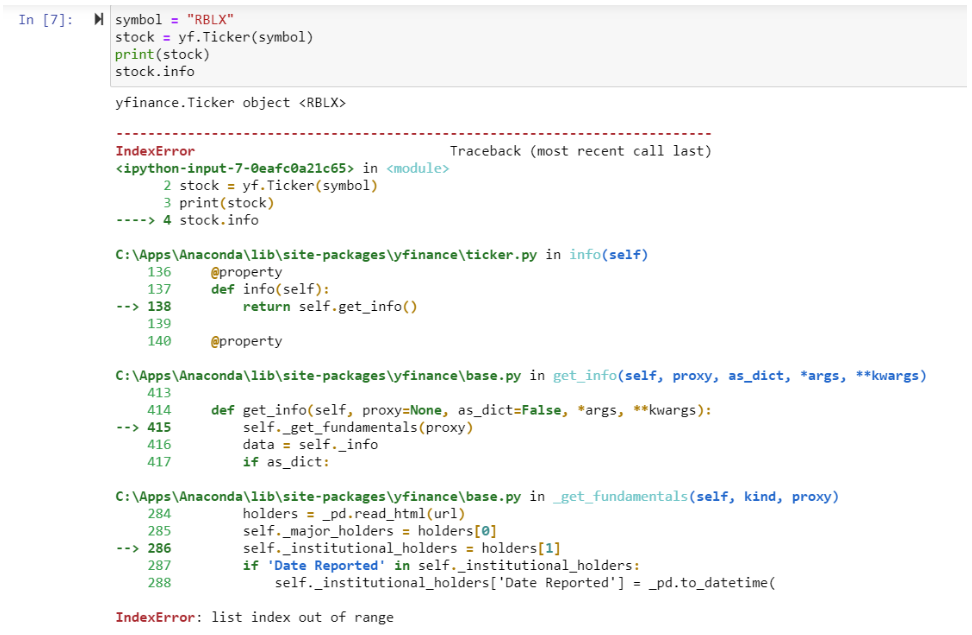


```
symbol = "AAPL"
stock = yf.Ticker(symbol)
print(stock)
stock.info
```

```
 'volume24Hr': None,
 'regularMarketDayHigh': 134.66,
 'navPrice': None,
 'averageDailyVolume10Day': 89822366,
 'totalAssets': None,
 'regularMarketPreviousClose': 131.24,
 'fiftyDayAverage': 123.43172,
 'trailingAnnualDividendRate': 0.807,
 'open': 132.44,
 'toCurrency': None,
 'averageVolume10days': 89822366,
 'expireDate': None,
 'yield': None,
 'algorithm': None,
 'dividendRate': 0.8200000000000001,
 'exDividendDate': 1612483200,
 'beta': 1.219525,
 'circulatingSupply': None,
 'startDate': None,
 'regularMarketDayLow': 131.94,
```

4. Invoking .info gives different keys depending on the type a ticker may represent. ETFs seem to have certain keys missing when compared with non-ETF tickers. A recent IPO company might return an empty result on invoking .info but return valid prices on invoking .history.
   Principles of concern: U2, M7

5. There are many cases where the API fails silently and sometimes expose internal errors (for example, Roblox, a company that IPO 2 weeks back, the API fails on invoking .info)
   Principles of concern: E3, Q5



```
In [7]:  symbol = "RBLX"
         stock = yf.Ticker(symbol)
         print(stock)
         stock.info
```

```
yfinance.Ticker object <RBLX>

---------------------------------------------------------------------------
IndexError                                Traceback (most recent call last)
<ipython-input-7-0eafc0a21c65> in <module>
      2 stock = yf.Ticker(symbol)
      3 print(stock)
----> 4 stock.info

C:\Apps\Anaconda\lib\site-packages\yfinance\ticker.py in info(self)
    136     @property
    137     def info(self):
--> 138         return self.get_info()
    139 
    140     @property

C:\Apps\Anaconda\lib\site-packages\yfinance\base.py in get_info(self, proxy, as_dict, *args, **kwargs)
    413 
    414     def get_info(self, proxy=None, as_dict=False, *args, **kwargs):
--> 415         self._get_fundamentals(proxy)
    416         data = self._info
    417         if as_dict:

C:\Apps\Anaconda\lib\site-packages\yfinance\base.py in _get_fundamentals(self, kind, proxy)
    284         holders = _pd.read_html(url)
    285         self._major_holders = holders[0]
--> 286         self._institutional_holders = holders[1]
    287         if 'Date Reported' in self._institutional_holders:
    288             self._institutional_holders['Date Reported'] = _pd.to_datetime(

IndexError: list index out of range
```

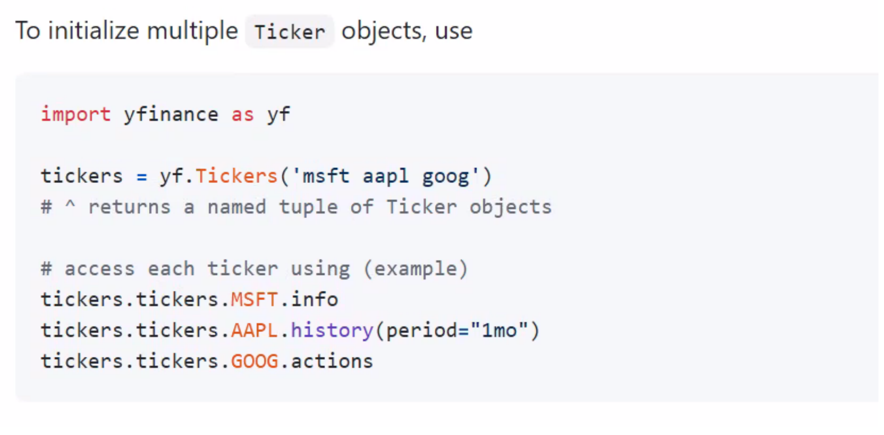
6. The API makes use of bad parameters as it takes the start date as String (YYYY-MM-DD) or _datetime. For start date, default is 1900-01-01 (String). In case of end date the default is “now”. Thus, resulting in unaccepted behaviors.
   Principles of concern: M3.1, M3.3, U3

7. Strange behavior in pdr.get_data_yahoo's start and end parameters. When a Pandas timestamp is passed to the parameter, the function does not fail, but produces inconsistent results. However, String parameters produce the correct result. (Issue: https://github.com/ranaroussi/yfinance/issues/679)
   Principles of concern: M3, U3

8. The history() and download() method has to take in a String period parameter or String date. Valid periods: 1d, 5d, 1mo, 3mo, 6mo, 1y, 2y, 5y, 10y, ytd, max. The format for the valid periods could be improved and extended to support a variety of formats. Valid periods should be more clearly defined and there could be improved to allow more flexible periods. Similarly, for intervals, the valid periods are valid intervals: 1m,2m,5m,15m,30m,60m,90m,1h,1d,5d,1wk,1mo,3mo. These are inconsistent with valid periods and have a high conceptual weight.
   Principles of Concern: M3, U8

9. Optional parameters like prepost, auto adjust, proxy, rounding are badly named. Some of these are booleans while others are strings. Some have a default value of False while others have a default value of True. Thus, resulting in inconsistencies and high complexity of use.
   Principles of Concern: M3, M4, M5

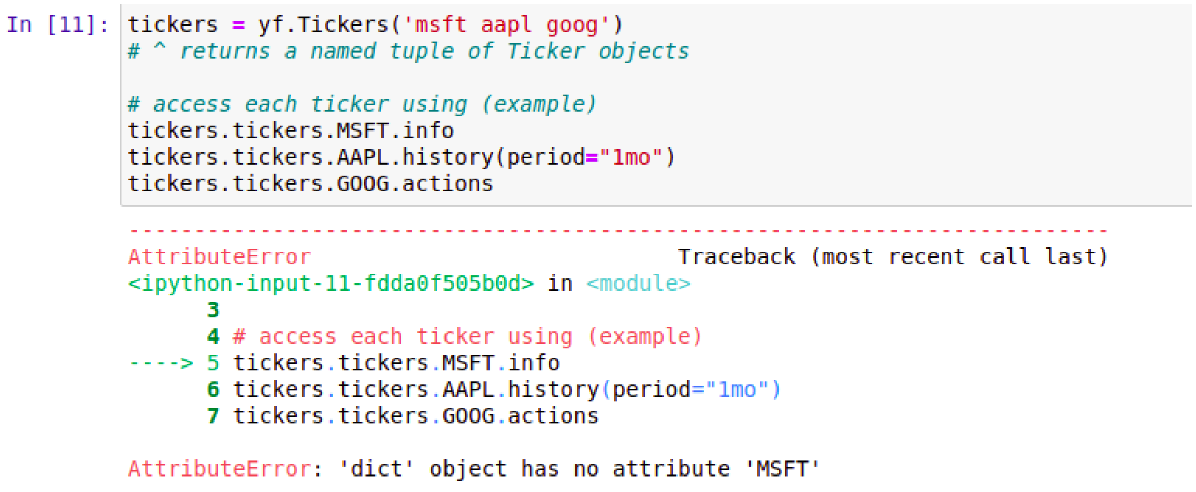
10. The API for initializing multiple tickers simultaneously appears to form a very non-elegant version of client code and has poor readability. It also prohibits dynamic instantiation and further usage.



To initialize multiple `Ticker` objects, use

```
import yfinance as yf

tickers = yf.Tickers('msft aapl goog')
# ^ returns a named tuple of Ticker objects

# access each ticker using (example)
tickers.tickers.MSFT.info
tickers.tickers.AAPL.history(period="1mo")
tickers.tickers.GOOG.actions
```

Additionally this is a case of bad documentation as a direct example from the documentation does not work as demonstrated for the multiple tickers example.



```
In [11]: tickers = yf.Tickers('msft aapl goog')
         # ^ returns a named tuple of Ticker objects

         # access each ticker using (example)
         tickers.tickers.MSFT.info
         tickers.tickers.AAPL.history(period="1mo")
         tickers.tickers.GOOG.actions
```

```
---------------------------------------------------------------------------
AttributeError                            Traceback (most recent call last)
<ipython-input-11-fdda0f505b0d> in <module>
      3
      4 # access each ticker using (example)
----> 5 tickers.tickers.MSFT.info
      6 tickers.tickers.AAPL.history(period="1mo")
      7 tickers.tickers.GOOG.actions

AttributeError: 'dict' object has no attribute 'MSFT'
```

Principles of Concern: R1

11. The API doesn't provide an option to set or change timezone. Normally a trader would be concerned about having data presented to them in either the local timezone or the timezone of the stock market where the stock is traded. The API should be able to get the user's default timezone and adjust the stock's data to match this timezone.
Principles of concern: U1, Q3

NEW API REQUIREMENTS

A detailed description and evolution of the requirements for the new API can be found in the requirements.md file.

First interview with Prof. Duane J. Seppi (BNY Mellon Professor of Finance at Tepper Business School, CMU) helped us identify the key securities that were of importance and lay down a skeleton of how the API abstraction would look like. Based on the discussion, we developed the requirement table consisting of requirements, its relative importance in API (out of 5), current status (if it is done: implemented, if it is pending: to be implemented OR if it is to be removed from requirements) and tracking ID for development purposes.

The second interview helped us in getting feedback on attributes and methods related to selected securities. Based on the interview we restructured our design to make the Base class more general and incorporate several popular technical indicators:

- Price indicators: Bollinger Bands, Moving Averages, Rate of change ratio, RSI, Linear Regression, Standard Deviation, Variance, Time Series Forecast
- Momentum Indicator: Balance Of Power, Commodity Channel Index
- Volume Indicator: Chaikin A/D Line

During the third interview, based on the feedback, we separated ETF from Stock class since certain information was not relevant for ETF (like company info, PE ratio etc). We also decided to remove the Commodities-futures security class and instead add a Miscellaneous class (to account for a security that's not stock, bond, ETF, mutual fund, currency, cryptocurrency). Another important decision based on common use cases and a responsible replacement for Tickers class was to add a Portfolio class to manage a group of Securities.

During the final interview we went over each of the security classes in detail and got feedback on improvements regarding what additional functionalities can be added to make the API more useful. The current yfinance API had limited functionalities and was very broken. This yayFinPy helps overcome those shortcomings by fixing the flaws and extending the utility of API by providing several helpful functions. In addition to the price, volume and momentum related technical indicators and the portfolio utility functionalities added during the prior interview stages, in this interview we were suggested several stock related functionalities to enhance the API. We concluded on adding functionalities like getting the news related to the stock, getting sentiment from public posts, getting tweets and providing additional stats about institutional holdings, major holders and mutual funds containing stocks.

FIXES AND IMPROVEMENTS:

- Revamped the API design entirely
- Gave structure and meaning to the API components which was completely missing
- Grouped common features into one Base class and used inheritance. It is easy to expand upon this API as base is very simple and logical, which works for most types of securities. The classes for supported securities are:
    - Stock()
    - MutualFund()
    - Currency()
    - TreasuryBonds()
    - ETF()
    - Misc()
    - Portfolio()
- Removed vague/irrelevant parameters for each type of security.
- Kept the functionalities for yfinance and added many new features.
- Made it easy for data analysis (its primary use case).
- Implemented the most common type of securities based on the inputs by our mentor.
- The rest are grouped under `Misc` class which is a simple wrapper to access yfinance objects and some extra methods which make it a valid security.
- Removed `Tickers()`, as it didn't make sense to just group some random ticker symbols in a group. Instead, made it a logical "portfolio" containing "security" objects which now is an entity describing the client's financial portfolio.
- Added `Portfolio` to manage group of securities. `Portfolio` object has value and returns associated with it. It provides several utility functions to work with the entire portfolio.
- Added custom `Exceptions` to help users pinpoint the real issue easily, give more information about the problem without revealing any API internal specifications and fail fast. The exceptions added are:
    - InputError

    - ParsingError
    - TwitterError
    - NewsError
    - SecurityTypeError
    - YfinanceError
- Added custom `Enumerations` instead of passing random Strings as parameters to many methods. Makes the client code more readable and reduces the possibility of the client making a mistake. Enums added were:
    - QuoteType:
        - CRYPTOCURRENCY = "CRYPTOCURRENCY"
        - CURRENCY = "CURRENCY"
        - EQUITY = "EQUITY"
        - ETF = "ETF"
        - INDEX = "INDEX"
        - MUTUALFUND = "MUTUALFUND"
        - MISC = "MISC"
    - Interval:
        - MINUTE_1 = "1m"
        - MINUTE_5 = "5m"
        - MINUTE_15 = "15m"
        - MINUTE_30 = "30m"
        - HOUR_1 = "1h"
        - DAY_1 = "1d"
        - DAY_5 = "5d"
        - WEEK_1 = "1wk"
        - MONTH_1 = "1mo"
        - MONTH_3 = "3mo"
    - Duration:
        - DAY_1 = "1d"
        - DAY_5 = "5d"
        - MONTH_1 = "1mo"
        - MONTH_3 = "3mo"
        - MONTH_6 = "6mo"
        - YEAR_1 = "1y"
        - YEAR_2 = "2y"
        - YEAR_5 = "5y"
        - YEAR_10 = "10y"
        - YEAR_TO_DATE = "ytd"
        - MAX = "max"
    - Multiplier:
        - ONCE = 1
        - TWICE = 2
        - THRICE = 3
        - QUADRICE = 4
        - HALF = 0.5
        - QUARTER = 0.25
- Make it easier for users to start using the API by providing clear documentation along with examples, unlike the original API even some of the examples from the documentation didn’t work.

API CODE COMPARISONS

Currency and Cryptocurrency

yfinance API code (old):

```
import yfinance as yf

currency = yf.Ticker("CNY=X")
currency_info = currency.info

if 'symbol' in currency_info.keys():
  # fetch price
  price = currency_info['regularMarketPrice']

  # fetch currency pair
  base_currency = ""
  if 'toCurrency' in currency_info.keys():
    if currency_info['toCurrency']:
      base_currency = currency_info['toCurrency'].split('=')[0]
    else:
      base_currency = currency_info['shortName'].split('/')[0]

  quote_currency = ""
  if 'fromCurrency' in currency_info.keys() and currency_info['fromCurrency']:
    quote_currency = currency_info['fromCurrency']
  elif 'currency' in currency_info.keys():
    quote_currency = currency_info['currency']

  print("Results: Price=%f, Base Currency=%s, Quote Currency=%s"
                        %(price, base_currency, quote_currency))
```

```
Results: Price=6.415600, Base Currency=USD, Quote Currency=CNY
```

yayFinPy API code (new):

```
from yayFinPy.currency import Currency

try:
  currency = Currency("CNY=X")
  # fetch price
  price = currency.price

  # fetch currency pair
  base_currency = currency.base_currency
  quote_currency = currency.quote_currency
  print("Results: Price=%f, Base Currency=%s, Quote Currency=%s"
                        %(price, base_currency, quote_currency))
except:
  print("Invalid Input Error!!!")
```

```
Results: Price=6.415600, Base Currency=USD, Quote Currency=CNY
```

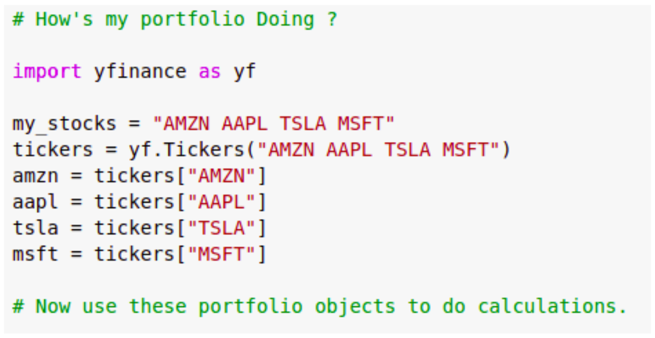
Currency is a unique type of security we are interested in. For currency and cryptocurrency, we are looking at a currency pair, which has a base currency and a quote currency. The price of the quote currency is the equivalence of 1 unit of the base currency.

The sample client code shows how to get this basic information using the original API and using our improved API. The old client code looks messy and complicated because it uses one method to get all information for currency, cryptos, and commodities, which returns an unstructured response that contains many irrelevant key-value pairs or null values. Also, it is worth mentioning that the naming of the keys in the original API response (e.g. fromCurrency, toCurrency) does not adhere to the domain convention.

In our new design, we addressed these problems and made the client code more concise and readable by restructuring the class design and attribute extractions.

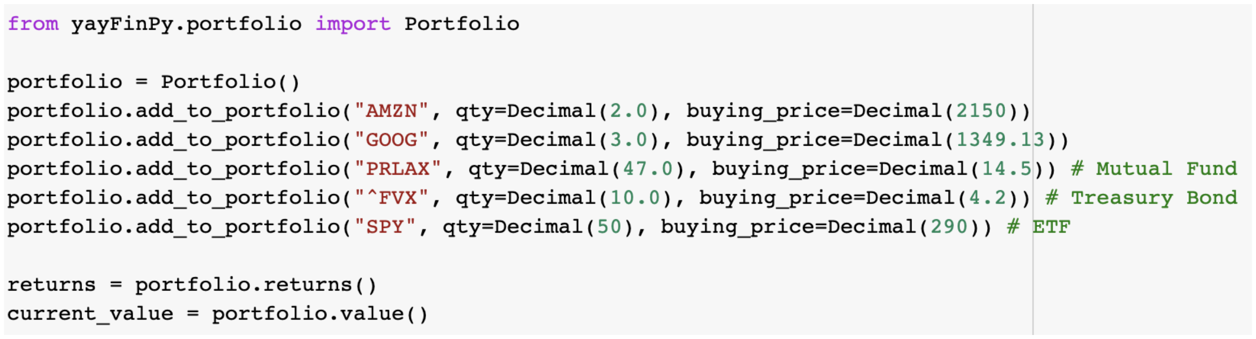
Portfolio

yfinance API code (old):



```python
# How's my portfolio Doing ?

import yfinance as yf

my_stocks = "AMZN AAPL TSLA MSFT"
tickers = yf.Tickers("AMZN AAPL TSLA MSFT")
amzn = tickers["AMZN"]
aapl = tickers["AAPL"]
tsla = tickers["TSLA"]
msft = tickers["MSFT"]

# Now use these portfolio objects to do calculations.
```

yayFinPy API code (new):



```python
from yayFinPy.portfolio import Portfolio

portfolio = Portfolio()
portfolio.add_to_portfolio("AMZN", qty=Decimal(2.0), buying_price=Decimal(2150))
portfolio.add_to_portfolio("GOOG", qty=Decimal(3.0), buying_price=Decimal(1349.13))
portfolio.add_to_portfolio("PRLAX", qty=Decimal(47.0), buying_price=Decimal(14.5)) # Mutual Fund
portfolio.add_to_portfolio("^FVX", qty=Decimal(10.0), buying_price=Decimal(4.2)) # Treasury Bond
portfolio.add_to_portfolio("SPY", qty=Decimal(50), buying_price=Decimal(290)) # ETF

returns = portfolio.returns()
current_value = portfolio.value()
```

Portfolio represents a collection of diverse securities that may be owned by an individual or an investment entity. A portfolio can consist of securities and their quantities, details like their value and how they perform and provide returns on the initial investment.

If a user of the API has a diverse set of securities and wants to analyze it to see how their portfolio is doing, the original yfinance API doesn't provide any way for users to do that. In yayFinPy, we provide a portfolio class exposed as a module to allow users to create a custom portfolio and update details on buying, as shown in the above sample code. The user can use the methods of this class to analyze the portfolio and look at the returns or current value. Users can add any of our defined securities to the portfolio.

Technical Analysis

yfinance API code (old):

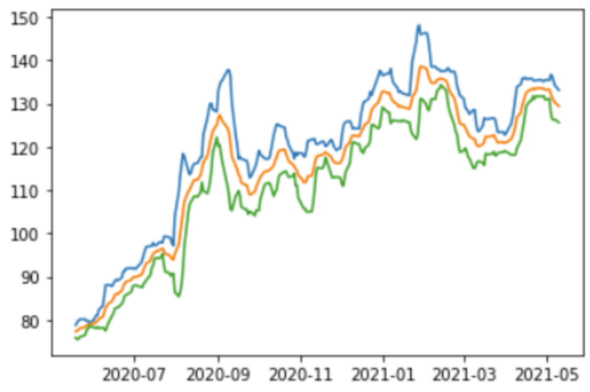
```
import statistics
data = apple.history(period="1y")
historical_price = data["Close"]
variance = []
moving_averages = []
window_size = 7
i = 0
while i < len(historical_price) - window_size + 1:
    this_window = historical_price[i : i + window_size]
    moving_averages.append(statistics.mean(this_window))
    variance.append(statistics.variance(this_window))
    i += 1
plt.plot(moving_averages)
plt.plot(variance)
```

yayFinPy API code (new):

```
apple_stock = Stock("AAPL")
# moving average
apple_stock.moving_average(Duration.YEAR_1)
# variance
apple_stock.variance(Duration.YEAR_1)
```

Supports several technical indicators e.g.

```
apple_stock.bollinger_bands(Duration.YEAR_1)
```



The yfinance API client code is really verbose since the client has to write the implementation to analyze the data. The new yayFinPy API provides several commonly used price, volume, and momentum technical indicators.

NEW FEATURES:

Based on our interactions with users in this domain, we found that one of the most popular use cases of the yfinance API is for analysis of financial information. In our redesigned API, we make it easier for users to perform data analysis by providing some common technical indicators and additional security specific methods for a better user experience.

Some of the new features we provide are:

- Get related news for a stock
- Sentiment Analysis of a stock
- Recent Tweets related to a stock
- Price, volume, and momentum based technical indicators.

RESULTS

In summary, there are several major improvements in our new API. First, we made the API easy to use by restructuring the class structure and providing approachable methods and properties related to the financial domain. Our design changes also enable users to write concise and readable client code for various financial analysis tasks. In addition to structural changes, our API introduces new features like Portfolio analysis and technical indicators for different securities. This also makes it much easier for users to use the API for more advanced tasks. Finally, our new API makes Portfolio interoperable with various Individual securities, which greatly improves the readability and usability of the class.

FUTURE WORK

Given the time constraint on this project, we only provide a fix for the API. There are ways we can make further improvements on the project. Some future work include: adding support for more specialized securities such as Commodities and adding more common operations typically used in a diverse Portfolio. We can also add the ability to create and test different investment strategies; for example, investing simulations and monte carlo simulations could be added based on the obtained data. It is also possible to add Machine Learning based sentiment analysis commonly used by the professional analysts.

PROBLEMS FACED AND LEARNINGS

Firstly, it was hard to find people working on the domain who were willing to help. After reaching out to multiple people, we received a response from Prof. Duane Seppi from the Tepper School of Business who was kind enough to provide his inputs, clarified our doubts regarding the finance concepts and provided valuable feedback on our API.

None of our team members were experts in the finance domain, and it was a task to understand the intricacies of the domain before we could start thinking about ways to improve the API. We learnt quite a lot about the world of finance and how a stock differs from a bond and now we can say we know enough to be able to distinguish between these securities and to be able to understand conversations about these topics.

We also learnt what an API designer faces in their day-to-day life with respect to feature requests, ease of use, et cetera. We learnt how to overcome those challenges and come up with something which is sensible to the user, adhering to the design principles we learnt in this course.

REFERENCES

[1] https://pypi.org/project/yfinance/

[2] https://corporatefinanceinstitute.com/resources/knowledge/finance/security

APPENDIX

The Code documentation can be found in the project repository:

https://github.com/cmu-api-design/Team13/tree/main/doc/yayFinPy

The Requirements document can be found in the project repository:

https://github.com/cmu-api-design/Team13/blob/main/requirements.md

The Design Rationale can be found in the project repository:

https://github.com/cmu-api-design/Team13/blob/main/design-rationale.md

The Work Attribution can be found in the project repository:

https://github.com/cmu-api-design/Team13/blob/main/work-attribution.md