# 2
# Market and Fundamental Data – Sources and Techniques

Data has always been an essential driver of trading, and traders have long made efforts to gain an advantage from access to superior information. These efforts date back at least to the rumors that the House of Rothschild benefited handsomely from bond purchases upon advance news about the British victory at Waterloo, which was carried by pigeons across the channel.

Today, investments in faster data access take the shape of the Go West consortium of leading **high-frequency trading (HFT)** firms that connects the **Chicago Mercantile Exchange (CME)** with Tokyo. The round-trip latency between the CME and the **BATS (Better Alternative Trading System)** exchanges in New York has dropped to close to the theoretical limit of eight milliseconds as traders compete to exploit arbitrage opportunities. At the same time, regulators and exchanges have started to introduce speed bumps that slow down trading to limit the adverse effects on competition of uneven access to information.

Traditionally, investors mostly relied on **publicly available market and fundamental data**. Efforts to create or acquire private datasets, for example, through proprietary surveys, were limited. Conventional strategies focus on equity fundamentals and build financial models on reported financials, possibly combined with industry or macro data to project earnings per share and stock prices. Alternatively, they leverage **technical analysis** to extract signals from market data using indicators computed from price and volume information.

**Machine learning (ML)** algorithms promise to exploit market and fundamental data more efficiently than human-defined rules and heuristics, particularly when combined with **alternative data**, which is the topic of the next chapter. We will illustrate how to apply ML algorithms ranging from linear models to **recurrent neural networks (RNNs)** to market and fundamental data and generate tradeable signals.

This chapter introduces market and fundamental data sources and explains how they reflect the environment in which they are created. The details of the **trading environment** matter not only for the proper interpretation of market data but also for the design and execution of your strategy and the implementation of realistic backtesting simulations.

We also illustrate how to access and work with trading and financial statement data from various sources using Python.

In particular, this chapter will cover the following topics:

- How market data reflects the structure of the trading environment
- Working with trade and quote data at minute frequency
- Reconstructing an order book from tick data using Nasdaq ITCH
- Summarizing tick data using various types of bars
- Working with **eXtensible Business Reporting Language** (**XBRL**)-encoded electronic filings
- Parsing and combining market and fundamental data to create a **price-to-earnings** (**P/E**) series
- How to access various market and fundamental data sources using Python

> You can find the code samples for this chapter and links to additional resources in the corresponding directory of the GitHub repository. The notebooks include color versions of the images.

# Market data reflects its environment

Market data is the product of how traders place orders for a financial instrument directly or through intermediaries on one of the numerous marketplaces, how they are processed, and how prices are set by matching demand and supply. As a result, the data reflects the institutional environment of trading venues, including the rules and regulations that govern orders, trade execution, and price formation. See Harris (2003) for a global overview and Jones (2018) for details on the U.S. market.

Algorithmic traders use algorithms, including ML, to analyze the flow of buy and sell orders and the resulting volume and price statistics to extract trade signals that capture insights into, for example, demand-supply dynamics or the behavior of certain market participants.

We will first review institutional features that impact the simulation of a trading strategy during a backtest before we start working with actual tick data created by one such environment, namely Nasdaq.

## Market microstructure – the nuts and bolts

Market microstructure studies how the **institutional environment** affects the trading process and shapes outcomes like price discovery, bid-ask spreads and quotes, intraday trading behavior, and transaction costs (Madhavan 2000; 2002). It is one of the fastest-growing fields of financial research, propelled by the rapid development of algorithmic and electronic trading.

Today, hedge funds sponsor in-house analysts to track the rapidly evolving, complex details and ensure execution at the best possible market prices and design strategies that exploit market frictions. We will provide only a brief overview of these key concepts before we dive into the data generated by trading. The references contain several sources that treat this subject in great detail.

## How to trade – different types of orders

Traders can place various types of buy or sell orders. Some orders guarantee immediate execution, while others may state a price threshold or other conditions that trigger execution. Orders are typically valid for the same trading day unless specified otherwise.

A *market order* is intended for immediate execution of the order upon arrival at the trading venue, at the price that prevails at that moment. In contrast, a *limit order* only executes if the market price is higher than the limit for a sell limit order, or lower than the limit for a buy limit order. A *stop order*, in turn, only becomes active when the market price rises above a specified price for a buy stop order, or falls below a specified price for a sell order. A *buy stop order* can be used to limit the losses of short sales. Stop orders may also have limits.

Numerous other conditions can be attached to orders. For example, *all or none orders* prevent partial execution; they are filled only if a specified number of shares is available and can be valid for a day or longer. They require special handling and are not visible to market participants. *Fill or kill orders* also prevent partial execution but cancel if not executed immediately. *Immediate or cancel orders* immediately buy or sell the number of shares that are available and cancel the remainder. *Not-held orders* allow the broker to decide on the time and price of execution. Finally, the market on *open/close orders* executes on or near the opening or closing of the market. Partial executions are allowed.

## Where to trade – from exchanges to dark pools

Securities trade in highly organized and **regulated exchanges** or with varying degrees of formality in **over-the-counter** (**OTC**) markets. An exchange is a central marketplace where buyers and sellers compete for the lowest ask and highest bid, respectively. Exchange regulations typically impose listing and reporting requirements to create transparency and attract more traders and liquidity. OTC markets, such as the Best Market (OTCQX) or the Venture Market (OTCQB), often have lower regulatory barriers. As a result, they are suitable for a far broader range of securities, including bonds or **American Depositary Receipts** (**ADRs**; equity listed on a foreign exchange, for example, for Nestlé, S.A.).

Exchanges may rely on bilateral trading or centralized order-driven systems that match all buy and sell orders according to certain rules. Many exchanges use intermediaries that provide liquidity by making markets in certain securities. These **intermediaries** include dealers that act as principals on their own behalf and brokers that trade as agents on behalf of others. **Price formation** may occur through auctions, such as in the **New York Stock Exchange** (**NYSE**), where the highest bid and lowest offer are matched, or through dealers who buy from sellers and sell to buyers.

Back in the day, companies either registered and traded mostly on the NYSE, or they traded on OTC markets like Nasdaq. On the NYSE, a sole **specialist** intermediated trades of a given security. The specialist received buy and sell orders via a broker and tracked limit orders in a central order book. Limit orders were executed with a priority based on

price and time. Buy market orders routed to the specialist transacted with the lowest ask (and sell market orders routed to the specialist transacted with the highest bid) in the limit order book, prioritizing earlier limit orders in the case of ties. Access to all orders in the central order book allowed the specialist to publish the best bid, ask prices, and set market prices based on the overall buy-sell imbalance.

On Nasdaq, multiple **market makers** facilitated stock trades. Each dealer provided their best bid and ask price to a central quotation system and stood ready to transact the specified number of shares at the specified prices. Traders would route their orders to the market maker with the best quote via their broker. The competition for orders made execution at fair prices very likely. Market makers ensured a fair and orderly market, provided liquidity, and disseminated prices like specialists but only had access to the orders routed to them as opposed to market-wide supply and demand. This fragmentation could create difficulties in identifying fair value market prices.

Today, **trading has fragmented**; instead of two principal venues in the US, there are more than thirteen displayed trading venues, including exchanges and (unregulated) **alternative trading systems (ATSs)** such as **electronic communication networks (ECNs)**. Each reports trades to the consolidated tape, but at different latencies. To make matters more difficult, the rules of engagement for each venue differ with several different pricing and queuing models.

The following table lists some of the larger global exchanges and the trading volumes for the 12 months ending 03/2018 in various asset classes, including derivatives. Typically, a minority of financial instruments account for most trading:

| | Stocks | | | | |
|---|---|---|---|---|---|
| Exchange | Market cap (USD mn) | # Listed companies | Volume / day (USD mn) | # Shares / day ('000) | # Options / day ('000) |
| NYSE | 23,138,626 | 2,294 | 78,410 | 6,122 | 1,546 |
| Nasdaq — US | 10,375,718 | 2,968 | 65,026 | 7,131 | 2,609 |
| Japan Exchange Group Inc. | 6,287,739 | 3,618 | 28,397 | 3,361 | 1 |
| Shanghai Stock Exchange | 5,022,691 | 1,421 | 34,736 | 9,801 | |

| | | | | | |
|---|---|---|---|---|---|
| Euronext | 4,649,073 | 1,240 | 9,410 | 836 | 304 |
| Hong Kong Exchanges and Clearing | 4,443,082 | 2,186 | 12,031 | 1,174 | 516 |
| LSE Group | 3,986,413 | 2,622 | 10,398 | 1,011 | |
| Shenzhen Stock Exchange | 3,547,312 | 2,110 | 40,244 | 14,443 | |
| Deutsche Boerse AG | 2,339,092 | 506 | 7,825 | 475 | |
| BSE India Limited | 2,298,179 | 5,439 | 602 | 1,105 | |
| National Stock Exchange of India Limited | 2,273,286 | 1,952 | 5,092 | 10,355 | |
| BATS Global Markets - US | | | | | 1,243 |
| Chicago Board Options Exchange | | | | | 1,811 |
| International Securities Exchange | | | | | 1,204 |

The ATSs mentioned previously include dozens of **dark pools** that allow traders to execute anonymously. They are estimated to account for 40 percent of all U.S. stock trades in 2017, compared with an estimated 16 percent in 2010. Dark pools emerged in the 1980s when the SEC allowed brokers to match buyers and sellers of big blocks of shares. The rise of high-frequency electronic trading and the 2007 SEC Order Protection rule that intended to spur competition and cut transaction costs through transparency as part of **Regulation National Market System (Reg NMS)** drove the growth of dark pools, as traders aimed to avoid the visibility of large trades (Mamudi 2017). Reg NMS also established the **National Best Bid and Offer (NBBO)** mandate for brokers to route orders to venues that offer the best price.

Some ATSs are called dark pools because they do not broadcast pre-trade data, including the presence, price, and amount of buy and sell orders as

traditional exchanges are required to do. However, dark pools report information about trades to the **Financial Industry Regulatory Authority** (**FINRA**) after they occur. As a result, dark pools do not contribute to the process of price discovery until after trade execution but provide protection against various HFT strategies outlined in the first chapter.

In the next section, we will see how market data captures trading activity and reflect the institutional infrastructure in U.S. markets.

# Working with high-frequency data

Two categories of market data cover the thousands of companies listed on U.S. exchanges that are traded under Reg NMS: the **consolidated feed** combines trade and quote data from each trading venue, whereas each individual exchange offers **proprietary products** with additional activity information for that particular venue.

In this section, we will first present proprietary order flow data provided by Nasdaq that represents the actual stream of orders, trades, and resulting prices as they occur on a tick-by-tick basis. Then, we will demonstrate how to regularize this continuous stream of data that arrives at irregular intervals into bars of a fixed duration. Finally, we will introduce AlgoSeek's equity minute bar data, which contains consolidated trade and quote information. In each case, we will illustrate how to work with the data using Python so that you can leverage these sources for your trading strategy.

## How to work with Nasdaq order book data

The **primary source of market data** is the order book, which updates in real time throughout the day to reflect all trading activity. Exchanges typically offer this data as a real-time service for a fee; however, they may provide some historical data for free.

In the United States, stock markets provide quotes in three tiers, namely Level L1, L2, and L3, that offer increasingly granular information and capabilities:

- **Level 1 (L1)**: Real-time bid- and ask-price information, as available from numerous online sources.
- **Level 2 (L2)**: Adds information about bid and ask prices by specific market makers as well as the size and time of recent transactions for better insights into the liquidity of a given equity.
- **Level 3 (L3)**: Adds the ability to enter or change quotes, execute orders, and confirm trades and is available only to market makers and exchange member firms. Access to Level 3 quotes permits registered brokers to meet best execution requirements.

The trading activity is reflected in numerous **messages about orders** sent by market participants. These messages typically conform to the **electronic Financial Information eXchange** (**FIX**) communications pro-

tocol for the real-time exchange of securities transactions and market data or a native exchange protocol.

## Communicating trades with the FIX protocol

Just like SWIFT is the message protocol for back-office (for example, in trade-settlement) messaging, the FIX protocol is the **de facto messaging standard** for communication before and during trade executions between exchanges, banks, brokers, clearing firms, and other market participants. Fidelity Investments and Salomon Brothers introduced FIX in 1992 to facilitate the electronic communication between broker-dealers and institutional clients who, until then, exchanged information over the phone.

It became popular in global equity markets before expanding into foreign exchange, fixed income and derivatives markets, and further into post-trade to support straight-through processing. Exchanges provide access to FIX messages as a real-time data feed that is **parsed by algorithmic traders** to track market activity and, for example, identify the footprint of market participants and anticipate their next move.

The sequence of messages allows for the **reconstruction of the order book**. The scale of transactions across numerous exchanges creates a large amount (~10 TB) of unstructured data that is challenging to process and, hence, can be a source of competitive advantage.

The FIX protocol, currently at version 5.0, is a free and open standard with a large community of affiliated industry professionals. It is self-describing, like the more recent XML, and a FIX session is supported by the underlying **Transmission Control Protocol** (**TCP**) layer. The community continually adds new functionality.

The protocol supports pipe-separated key-value pairs, as well as a **tag-based FIXML** syntax. A sample message that requests a server login would look as follows:

```
8=FIX.5.0|9=127|35=A|59=theBroker.123456|56=CSERVER|34=1|32=20180117- 08:03:04|57=TRADE|50=any_str
```

There are a few open source FIX implementations in Python that can be used to formulate and parse FIX messages. The service provider Interactive Brokers offers a FIX-based **computer-to-computer interface** (**CTCI**) for automated trading (refer to the resources section for this chapter in the GitHub repository).

## The Nasdaq TotalView-ITCH data feed

While FIX has a dominant market share, exchanges also offer native protocols. Nasdaq offers a TotalView-ITCH **direct data-feed protocol**, which allows subscribers to **track individual orders** for equity instruments from placement to execution or cancellation.

Historical records of this data flow permit the reconstruction of the order book that keeps track of the active limit orders for a specific security. The

order book reveals the **market depth** throughout the day by listing the number of shares being bid or offered at each price point. It may also identify the market participant responsible for specific buy and sell orders unless they are placed anonymously. Market depth is a key indicator of liquidity and the **potential price impact** of sizable market orders.

In addition to matching market and limit orders, Nasdaq also operates **auctions or crosses** that execute a large number of trades at market opening and closing. Crosses are becoming more important as passive investing continues to grow and traders look for opportunities to execute larger blocks of stock. TotalView also disseminates the **Net Order Imbalance Indicator** (**NOII**) for Nasdaq opening and closing crosses and Nasdaq IPO/Halt Cross.

### How to parse binary order messages

The ITCH v5.0 specification declares over 20 message types related to system events, stock characteristics, the placement and modification of limit orders, and trade execution. It also contains information about the net order imbalance before the open and closing cross.

Nasdaq offers samples of daily binary files for several months. The GitHub repository for this chapter contains a notebook, `parse_itch_order_flow_messages.ipynb`, that illustrates how to download and parse a sample file of ITCH messages. The notebook `rebuild_nasdaq_order_book.ipynb` then goes on to reconstruct both the executed trades and the order book for any given ticker.

The following table shows the frequency of the **most common message types** for the sample file date October 30, 2019:

| Message type | Order book impact | Number of messages |
|---|---|---|
| A | New unattributed limit order | 127,214,649 |
| D | Order canceled | 123,296,742 |
| U | Order canceled and replaced | 25,513,651 |
| E | Full or partial execution; possibly multiple messages for the same original order | 7,316,703 |
| X | Modified after partial cancellation | 3,568,735 |
| F | Add attributed order | 1,423,908 |
| P | Trade message (non-cross) | 1,525,363 |
| C | Executed in whole or in part at a price different from the initial dis- | 129,729 |

| | | |
|---|---|---|
| | play price | |
| Q | Cross trade message | 17,775 |

For each message, the **specification** lays out the components and their respective length and data types:

| Name | Offset | Length | Value | Notes |
|---|---|---|---|---|
| Message type | 0 | 1 | S | System event message. |
| Stock locate | 1 | 2 | Integer | Always 0. |
| Tracking number | 3 | 2 | Integer | Nasdaq internal tracking number. |
| Timestamp | 5 | 6 | Integer | The number of nanoseconds since midnight. |
| Order reference number | 11 | 8 | Integer | The unique reference number assigned to the new order at the time of receipt. |
| Buy/sell indicator | 19 | 1 | Alpha | The type of order being added: B = Buy Order, and S = Sell Order. |
| Shares | 20 | 4 | Integer | The total number of shares associated with the order being added to the book. |
| Stock | 24 | 8 | Alpha | Stock symbol, right-padded with spaces. |
| Price | 32 | 4 | Price (4) | The display price of the new order. Refer to *Data Types* in the specification for field processing notes. |
| Attribution | 36 | 4 | Alpha | The Nasdaq market participant identifier associated with the entered order. |

Python provides the `struct` module to parse binary data using format strings that identify the message elements by indicating the length and type of the various components of the `byte` string as laid out in the specification.

Let's walk through the critical steps required to parse the trading messages and reconstruct the order book:

1. The ITCH parser relies on the message specifications provided in the file `message_types.xlsx` (refer to the notebook `parse_itch_order_flow_messages.ipynb` for details). It assembles format strings according to the `formats` dictionary:

```
formats = {
    ('integer', 2): 'H',  # int of length 2 => format string 'H'
    ('integer', 4): 'I',
    ('integer', 6): '6s', # int of length 6 => parse as string,
      convert later
    ('integer', 8): 'Q',
    ('alpha', 1)  : 's',
    ('alpha', 2)  : '2s',
    ('alpha', 4)  : '4s',
    ('alpha', 8)  : '8s',
    ('price_4', 4): 'I',
    ('price_8', 8): 'Q',
}
```

2. The parser translates the message specs into format strings and named tuples that capture the message content:

```
# Get ITCH specs and create formatting (type, length) tuples
specs = pd.read_csv('message_types.csv')
specs['formats'] = specs[['value', 'length']].apply(tuple,
                           axis=1).map(formats)
# Formatting for alpha fields
alpha_fields = specs[specs.value == 'alpha'].set_index('name')
alpha_msgs = alpha_fields.groupby('message_type')
alpha_formats = {k: v.to_dict() for k, v in alpha_msgs.formats}
alpha_length = {k: v.add(5).to_dict() for k, v in alpha_msgs.length}
# Generate message classes as named tuples and format strings
message_fields, fstring = {}, {}
for t, message in specs.groupby('message_type'):
    message_fields[t] = namedtuple(typename=t,
                                   field_names=message.name.tolist())
    fstring[t] = '>' + ''.join(message.formats.tolist())
```

3. Fields of the alpha type require postprocessing, as defined in the `format_alpha` function:

```
def format_alpha(mtype, data):
    """Process byte strings of type alpha"""
    for col in alpha_formats.get(mtype).keys():
        if mtype != 'R' and col == 'stock':
            data = data.drop(col, axis=1)
            continue
```

```
            data.loc[:, col] = (data.loc[:, col]
                                .str.decode("utf-8")
                                .str.strip())
            if encoding.get(col):
                data.loc[:, col] = data.loc[:, col].map(encoding.get(col))
        return data
```

The binary file for a single day contains over 300,000,000 messages that are worth over 9 GB. The script appends the parsed result iteratively to a file in the fast HDF5 format to avoid memory constraints. (Refer to the *Efficient data storage with pandas* section later in this chapter for more information on the HDF5 format.)

The following (simplified) code processes the binary file and produces the parsed orders stored by message type:

```
with (data_path / file_name).open('rb') as data:
    while True:
        message_size = int.from_bytes(data.read(2), byteorder='big',
                       signed=False)
        message_type = data.read(1).decode('ascii')
        message_type_counter.update([message_type])
        record = data.read(message_size - 1)
        message = message_fields[message_type]._make(
            unpack(fstring[message_type], record))
        messages[message_type].append(message)

        # deal with system events like market open/close
        if message_type == 'S':
            timestamp = int.from_bytes(message.timestamp,
                                       byteorder='big')
            if message.event_code.decode('ascii') == 'C': # close
                store_messages(messages)
                break
```

### Summarizing the trading activity for all 8,500 stocks

As expected, a small number of the 8,500-plus securities traded on this day account for most trades:

```
with pd.HDFStore(itch_store) as store:
    stocks = store['R'].loc[:, ['stock_locate', 'stock']]
    trades = (store['P'].append(
            store['Q'].rename(columns={'cross_price': 'price'}),
            sort=False).merge(stocks))
trades['value'] = trades.shares.mul(trades.price)
trades['value_share'] = trades.value.div(trades.value.sum())
trade_summary = (trades.groupby('stock').value_share
                 .sum().sort_values(ascending=False))
trade_summary.iloc[:50].plot.bar(figsize=(14, 6),
                                 color='darkblue',
                                 title='Share of Traded Value')
f = lambda y, _: '{:.0%}'.format(y)
plt.gca().yaxis.set_major_formatter(FuncFormatter(f))
```

*Figure 2.1* shows the resulting plot:

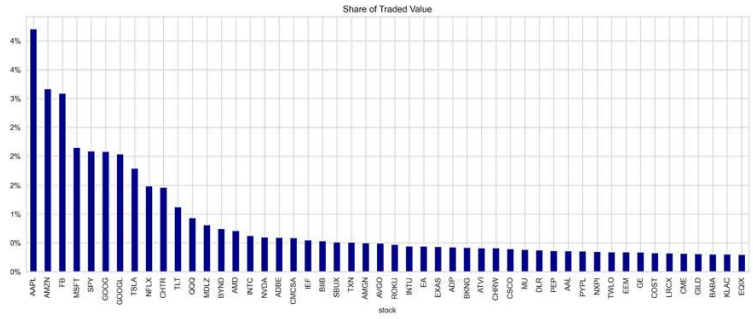


Figure 2.1: The share of traded value of the 50 most traded securities

### How to reconstruct all trades and the order book

The parsed messages allow us to rebuild the order flow for the given day. The `'R'` message type contains a listing of all stocks traded during a given day, including information about **initial public offerings** (**IPOs**) and trading restrictions.

Throughout the day, new orders are added, and orders that are executed and canceled are removed from the order book. The proper accounting for messages that reference orders placed on a prior date would require tracking the order book over multiple days.

The `get_messages()` function illustrates how to collect the orders for a single stock that affects trading. (Refer to the ITCH specification for details about each message.) The code is slightly simplified; refer to the notebook `rebuild_nasdaq_order_book.ipynb` for further details:

```
def get_messages(date, stock=stock):
    """Collect trading messages for given stock"""
    with pd.HDFStore(itch_store) as store:
        stock_locate = store.select('R', where='stock =
                                    stock').stock_locate.iloc[0]
        target = 'stock_locate = stock_locate'
        data = {}
        # relevant message types
        messages = ['A', 'F', 'E', 'C', 'X', 'D', 'U', 'P', 'Q']
        for m in messages:
            data[m] = store.select(m,
              where=target).drop('stock_locate', axis=1).assign(type=m)
    order_cols = ['order_reference_number', 'buy_sell_indicator',
                  'shares', 'price']
    orders = pd.concat([data['A'], data['F']], sort=False,
                       ignore_index=True).loc[:, order_cols]
    for m in messages[2: -3]:
        data[m] = data[m].merge(orders, how='left')
    data['U'] = data['U'].merge(orders, how='left',
                                right_on='order_reference_number',
                                left_on='original_order_reference_number',
                                suffixes=['', '_replaced'])
    data['Q'].rename(columns={'cross_price': 'price'}, inplace=True)
    data['X']['shares'] = data['X']['cancelled_shares']
    data['X'] = data['X'].dropna(subset=['price'])
    data = pd.concat([data[m] for m in messages], ignore_index=True,
                     sort=False)
```

Reconstructing successful trades—that is, orders that were executed as opposed to those that were canceled from trade-related message types `C`, `E`, `P`, and `Q`—is relatively straightforward:

```
def get_trades(m):
    """Combine C, E, P and Q messages into trading records"""
    trade_dict = {'executed_shares': 'shares', 'execution_price': 'price'}
    cols = ['timestamp', 'executed_shares']
    trades = pd.concat([m.loc[m.type == 'E',
                              cols + ['price']].rename(columns=trade_dict),
                        m.loc[m.type == 'C',
                              cols + ['execution_price']]
                        .rename(columns=trade_dict),
                        m.loc[m.type == 'P', ['timestamp', 'price',
                                              'shares']],
                        m.loc[m.type == 'Q',
                              ['timestamp', 'price', 'shares']]
                        .assign(cross=1), ],
                       sort=False).dropna(subset=['price']).fillna(0)
    return trades.set_index('timestamp').sort_index().astype(int)
```

The order book keeps track of limit orders, and the various price levels for buy and sell orders constitute the depth of the order book. Reconstructing the order book for a given level of depth requires the following steps:

The `add_orders()` function accumulates sell orders in ascending order and buy orders in descending order for a given timestamp up to the desired level of depth:

```
def add_orders(orders, buysell, nlevels):
    new_order = []
    items = sorted(orders.copy().items())
    if buysell == 1:
        items = reversed(items)
    for i, (p, s) in enumerate(items, 1):
        new_order.append((p, s))
        if i == nlevels:
            break
    return orders, new_order
```

We iterate over all ITCH messages and process orders and their replacements as required by the specification:

```
for message in messages.itertuples():
    i = message[0]
    if np.isnan(message.buy_sell_indicator):
        continue
    message_counter.update(message.type)
    buysell = message.buy_sell_indicator
    price, shares = None, None
    if message.type in ['A', 'F', 'U']:
        price, shares = int(message.price), int(message.shares)
        current_orders[buysell].update({price: shares})
        current_orders[buysell], new_order =
          add_orders(current_orders[buysell], buysell, nlevels)
```

```
            order_book[buysell][message.timestamp] = new_order
    if message.type in ['E', 'C', 'X', 'D', 'U']:
        if message.type == 'U':
            if not np.isnan(message.shares_replaced):
                price = int(message.price_replaced)
                shares = -int(message.shares_replaced)
        else:
            if not np.isnan(message.price):
                price = int(message.price)
                shares = -int(message.shares)
        if price is not None:
            current_orders[buysell].update({price: shares})
            if current_orders[buysell][price] <= 0:
                current_orders[buysell].pop(price)
            current_orders[buysell], new_order =
              add_orders(current_orders[buysell], buysell, nlevels)
            order_book[buysell][message.timestamp] = new_order
```

*Figure 2.2* highlights the depth of liquidity at any given point in time using different intensities that visualize the number of orders at different price levels. The left panel shows how the distribution of limit order prices was weighted toward buy orders at higher prices.

The right panel plots the evolution of limit orders and prices throughout the trading day: the dark line tracks the prices for executed trades during market hours, whereas the red and blue dots indicate individual limit orders on a per-minute basis (refer to the notebook for details):

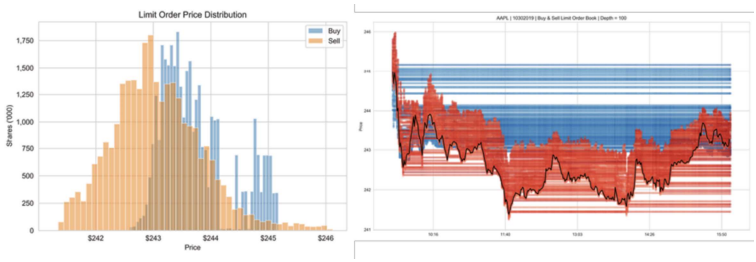


Figure 2.2: AAPL market liquidity according to the order book

## From ticks to bars – how to regularize market data

The trade data is indexed by nanoseconds, arrives at irregular intervals, and is very noisy. The **bid-ask bounce**, for instance, causes the price to oscillate between the bid and ask prices when trade initiation alternates between buy and sell market orders. To improve the noise-signal ratio and the statistical properties of the price series, we need to resample and regularize the tick data by aggregating the trading activity.

We typically collect the **open (first), high, low, and closing (last) price and volume** (jointly abbreviated as **OHLCV**) for the aggregated period, alongside the **volume-weighted average price** (**VWAP**) and the timestamp associated with the data.

Refer to the `normalize_tick_data.ipynb` notebook in the folder for this chapter on GitHub for additional details.

### The raw material – tick bars

The following code generates a plot of the raw tick price and volume data for AAPL:

```
stock, date = 'AAPL', '20191030'
title = '{} | {}'.format(stock, pd.to_datetime(date).date()
with pd.HDFStore(itch_store) as store:
    sys_events = store['S'].set_index('event_code') # system events
    sys_events.timestamp = sys_events.timestamp.add(pd.to_datetime(date)).dt.time
    market_open = sys_events.loc['Q', 'timestamp']
    market_close = sys_events.loc['M', 'timestamp']
with pd.HDFStore(stock_store) as store:
    trades = store['{}/trades'.format(stock)].reset_index()
trades = trades[trades.cross == 0] # excluding data from open/close crossings
trades.price = trades.price.mul(1e-4) # format price
trades = trades[trades.cross == 0]    # exclude crossing trades
trades = trades.between_time(market_open, market_close) # market hours only
tick_bars = trades.set_index('timestamp')
tick_bars.index = tick_bars.index.time
tick_bars.price.plot(figsize=(10, 5), title=title), lw=1)
```

*Figure 2.3* displays the resulting plot:

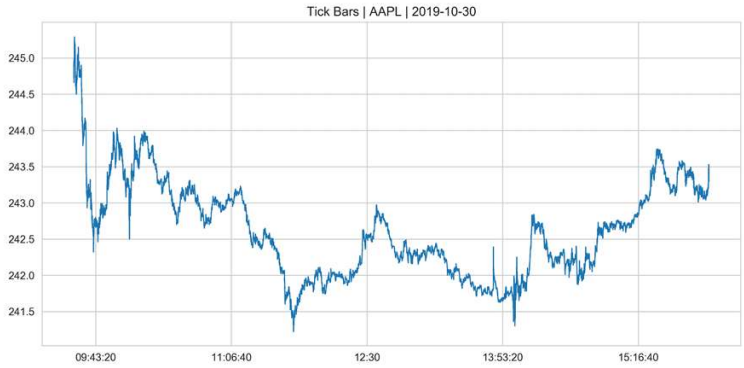


Figure 2.3: Tick bars

The tick returns are far from normally distributed, as evidenced by the low p-value of `scipy.stats.normaltest`:

```
from scipy.stats import normaltest
normaltest(tick_bars.price.pct_change().dropna())
NormaltestResult(statistic=62408.76562431228, pvalue=0.0)
```

### Plain-vanilla denoising – time bars

Time bars involve trade aggregation by period. The following code gets the data for the time bars:

```
def get_bar_stats(agg_trades):
    vwap = agg_trades.apply(lambda x: np.average(x.price,
           weights=x.shares)).to_frame('vwap')
    ohlc = agg_trades.price.ohlc()
```

```
    vol = agg_trades.shares.sum().to_frame('vol')
    txn = agg_trades.shares.size().to_frame('txn')
    return pd.concat([ohlc, vwap, vol, txn], axis=1)
resampled = trades.groupby(pd.Grouper(freq='1Min'))
time_bars = get_bar_stats(resampled)
```

We can display the result as a price-volume chart:

```
def price_volume(df, price='vwap', vol='vol', suptitle=title, fname=None):
    fig, axes = plt.subplots(nrows=2, sharex=True, figsize=(15, 8))
    axes[0].plot(df.index, df[price])
    axes[1].bar(df.index, df[vol], width=1 / (len(df.index)),
                color='r')
    xfmt = mpl.dates.DateFormatter('%H:%M')
    axes[1].xaxis.set_major_locator(mpl.dates.HourLocator(interval=3))
    axes[1].xaxis.set_major_formatter(xfmt)
    axes[1].get_xaxis().set_tick_params(which='major', pad=25)
    axes[0].set_title('Price', fontsize=14)
    axes[1].set_title('Volume', fontsize=14)
    fig.autofmt_xdate()
    fig.suptitle(suptitle)
    fig.tight_layout()
    plt.subplots_adjust(top=0.9)
price_volume(time_bars)
```

The preceding code produces *Figure 2.4*:


Figure 2.4: Time bars

Alternatively, we can represent the data as a candlestick chart using the Bokeh plotting library:

```
resampled = trades.groupby(pd.Grouper(freq='5Min')) # 5 Min bars for better print
df = get_bar_stats(resampled)
increase = df.close > df.open
decrease = df.open > df.close
w = 2.5 * 60 * 1000 # 2.5 min in ms
WIDGETS = "pan, wheel_zoom, box_zoom, reset, save"
p = figure(x_axis_type='datetime', tools=WIDGETS, plot_width=1500,
           title = "AAPL Candlestick")
p.xaxis.major_label_orientation = pi/4
p.grid.grid_line_alpha=0.4
p.segment(df.index, df.high, df.index, df.low, color="black")
```

```
p.vbar(df.index[increase], w, df.open[increase], df.close[increase],
       fill_color="#D5E1DD", line_color="black")
p.vbar(df.index[decrease], w, df.open[decrease], df.close[decrease],
       fill_color="#F2583E", line_color="black")
show(p)
```

This produces the plot in *Figure 2.5*:

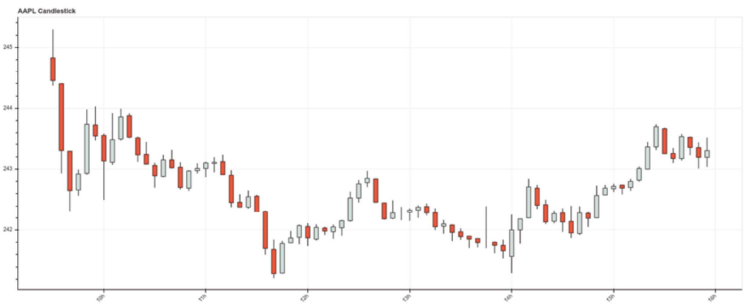


Figure 2.5: Bokeh candlestick plot

### Accounting for order fragmentation – volume bars

Time bars smooth some of the noise contained in the raw tick data but may fail to account for the fragmentation of orders. Execution-focused algorithmic trading may aim to match the **volume-weighted average price** (**VWAP**) over a given period. This will divide a single order into multiple trades and place orders according to historical patterns. Time bars would treat the same order differently, even though no new information has arrived in the market.

Volume bars offer an alternative by aggregating trade data according to volume. We can accomplish this as follows:

```
min_per_trading_day = 60 * 7.5
trades_per_min = trades.shares.sum() / min_per_trading_day
trades['cumul_vol'] = trades.shares.cumsum()
df = trades.reset_index()
by_vol = (df.groupby(df.cumul_vol.
                     div(trades_per_min)
                     .round().astype(int)))
vol_bars = pd.concat([by_vol.timestamp.last().to_frame('timestamp'),
                      get_bar_stats(by_vol)], axis=1)
price_volume(vol_bars.set_index('timestamp'))
```

We get the plot in *Figure 2.6* for the preceding code:


Figure 2.6: Volume bars

### Accounting for price changes – dollar bars

When asset prices change significantly, or after stock splits, the value of a given amount of shares changes. Volume bars do not correctly reflect this and can hamper the comparison of trading behavior for different periods that reflect such changes. In these cases, the volume bar method should be adjusted to utilize the product of shares and prices to produce dollar bars.

The following code shows the computation for dollar bars:

```
value_per_min = trades.shares.mul(trades.price).sum()/(60*7.5) # min per trading day
trades['cumul_val'] = trades.shares.mul(trades.price).cumsum()
df = trades.reset_index()
by_value = df.groupby(df.cumul_val.div(value_per_min).round().astype(int))
dollar_bars = pd.concat([by_value.timestamp.last().to_frame('timestamp'), get_bar_stats(by_value)]
price_volume(dollar_bars.set_index('timestamp'),
             suptitle=f'Dollar Bars | {stock} | {pd.to_datetime(date).date()}')
```

The plot looks quite similar to the volume bar since the price has been fairly stable throughout the day:


Figure 2.7: Dollar bars

## AlgoSeek minute bars – equity quote and trade data

AlgoSeek provides historical intraday data of the quality previously available only to institutional investors. The AlgoSeek Equity bars provide very detailed intraday quote and trade data in a user-friendly format, which is aimed at making it easy to design and backtest intraday ML-driven strategies. As we will see, the data includes not only OHLCV information but also information on the bid-ask spread and the number of ticks with up and down price moves, among others.

AlgoSeek has been so kind as to provide samples of minute bar data for the Nasdaq 100 stocks from 2013-2017 for demonstration purposes and will make a subset of this data available to readers of this book.

In this section, we will present the available trade and quote information and show how to process the raw data. In later chapters, we will demonstrate how you can use this data for ML-driven intraday strategies.

### From the consolidated feed to minute bars

AlgoSeek minute bars are based on data provided by the **Securities Information Processor** (**SIP**), which manages the consolidated feed mentioned at the beginning of this section. You can view the documentation at `https://www.algoseek.com/samples/`.

The SIP aggregates the best bid and offers quotes from each exchange, as well as the resulting trades and prices. Exchanges are prohibited by law from sending their quotes and trades to direct feeds before sending them to the SIP. Given the fragmented nature of U.S. equity trading, the consolidated feed provides a convenient snapshot of the current state of the market.

More importantly, the SIP acts as the benchmark used by regulators to determine the **National Best Bid and Offer** (**NBBO**) according to Reg NMS. The OHLC bar quote prices are based on the NBBO, and each bid or ask quote price refers to an NBBO price.

Every exchange publishes its top-of-book price and the number of shares available at that price. The NBBO changes when a published quote improves the NBBO. Bid/ask quotes persist until there is a change due to trade, price improvement, or the cancelation of the latest bid or ask. While historical OHLC bars are often based on trades during the bar period, NBBO bid/ask quotes may be carried forward from the previous bar until there is a new NBBO event.

AlgoSeek bars cover the whole trading day, from the opening of the first exchange until the closing of the last exchange. Bars outside regular market hours normally exhibit limited activity. Trading hours, in Eastern Time, are:

- Premarket: Approximately 04:00:00 (this varies by exchange) to 09:29:59
- Market: 09:30:00 to 16:00:00
- Extended hours: 16:00:01 to 20:00:00

### Quote and trade data fields

The minute bar data contains up to 54 fields. There are eight fields for the **open**, **high**, **low**, and **close** elements of the bar, namely:

- The timestamp for the bar and the corresponding trade
- The price and the size for the prevailing bid-ask quote and the relevant trade

There are also 14 data points with **volume information** for the bar period:

- The number of shares and corresponding trades
- The trade volumes at or below the bid, between the bid quote and the midpoint, at the midpoint, between the midpoint and the ask quote, and at or above the ask, as well as for crosses
- The number of shares traded with upticks or downticks, that is, when the price rose or fell, as well as when the price did not change, differentiated by the previous direction of price movement

The AlgoSeek data also contains the number of shares **reported to FINRA** and processed internally at broker-dealers, by dark pools, or OTC. These trades represent volume that is hidden or not publicly available until after the fact.

Finally, the data includes the **volume-weighted average price** (**VWAP**) and minimum and maximum bid-ask spread for the bar period.

### How to process AlgoSeek intraday data

In this section, we'll process the AlgoSeek sample data. The `data` directory on GitHub contains instructions on how to download that data from AlgoSeek.

The minute bar data comes in four versions: with and without quote information, and with or without FINRA's reported volume. There is one zipped folder per day, containing one CSV file per ticker.

The following code example extracts the trade-only minute bar data into daily `.parquet` files:

```
directories = [Path(d) for d in ['1min_trades']]
target = directory / 'parquet'
for zipped_file in directory.glob('*/**/*.zip'):
    fname = zipped_file.stem
    print('\t', fname)
    zf = ZipFile(zipped_file)
    files = zf.namelist()
    data = (pd.concat([pd.read_csv(zf.open(f),
                                   parse_dates=[['Date',
                                                 'TimeBarStart']])
                       for f in files],
                      ignore_index=True)
            .rename(columns=lambda x: x.lower())
            .rename(columns={'date_timebarstart': 'date_time'})
            .set_index(['ticker', 'date_time']))
    data.to_parquet(target / (fname + '.parquet'))
```

We can combine the `parquet` files into a single piece of HDF5 storage as follows, yielding 53.8 million records that consume 3.2 GB of memory and covering 5 years and 100 stocks:

```
path = Path('1min_trades/parquet')
df = pd.concat([pd.read_parquet(f) for f in path.glob('*.parquet')]).dropna(how='all', axis=1)
df.columns = ['open', 'high', 'low', 'close', 'trades', 'volume', 'vwap']
df.to_hdf('data.h5', '1min_trades')
print(df.info(null_counts=True))
MultiIndex: 53864194 entries, (AAL, 2014-12-22 07:05:00) to (YHOO, 2017-06-16 19:59:00)
Data columns (total 7 columns):
open      53864194 non-null float64
high      53864194 non-null float64
Low       53864194 non-null float64
close     53864194 non-null float64
trades    53864194 non-null int64
volume    53864194 non-null int64
vwap      53852029 non-null float64
```

We can use `plotly` to quickly create an interactive candlestick plot for one day of AAPL data to view in a browser:

```
idx = pd.IndexSlice
with pd.HDFStore('data.h5') as store:
    print(store.info())
    df = (store['1min_trades']
          .loc[idx['AAPL', '2017-12-29'], :]
          .reset_index())
fig = go.Figure(data=go.Ohlc(x=df.date_time,
                             open=df.open,
                             high=df.high,
                             low=df.low,
                             close=df.close))
```

*Figure 2.8* shows the resulting static image:


Figure 2.8: Plotly candlestick plot

AlgoSeek also provides adjustment factors to correct pricing and volumes for stock splits, dividends, and other corporate actions.

# API access to market data

There are several options you can use to access market data via an API using Python. We will first present a few sources built into the pandas library and the `yfinance` tool that facilitates the downloading of end-of-day market data and recent fundamental data from Yahoo! Finance.

Then we will briefly introduce the trading platform Quantopian, the data provider Quandl, and the Zipline backtesting library that we will use later in the book, as well as listing several additional options to access various types of market data. The directory `data_providers` on GitHub contains several notebooks that illustrate the usage of these options.

## Remote data access using pandas

The pandas library enables access to data displayed on websites using the `read_html` function and access to the API endpoints of various data providers through the related `pandas-datareader` library.

### Reading HTML tables

Downloading the content of one or more HTML tables, such as for the constituents of the S&P 500 index from Wikipedia, works as follows:

```
sp_url = 'https://en.wikipedia.org/wiki/List_of_S%26P_500_companies'
sp = pd.read_html(sp_url, header=0)[0] # returns a list for each table
sp.info()
RangeIndex: 505 entries, 0 to 504
Data columns (total 9 columns):
Symbol                   505 non-null object
Security                 505 non-null object
SEC filings               505 non-null object
GICS Sector              505 non-null object
GICS Sub Industry        505 non-null object
Headquarters Location    505 non-null object
Date first added          408 non-null object
CIK                      505 non-null int64
Founded                  234 non-null object
```

### pandas-datareader for market data

pandas used to facilitate access to data provider APIs directly, but this functionality has moved to the `pandas-datareader` library (refer to the `README` for links to the documentation).

The stability of the APIs varies with provider policies and continues to change. Please consult the documentation for up-to-date information. As of December 2019, at version 0.8.1, the following sources are available:

| Source | Scope | Comment |
|---|---|---|
| Tiingo | Historical end-of-day prices on equities, mutual funds, | Free registration for the API key. Free ac- |

| | | |
|---|---|---|
| | and ETF. | counts can access only 500 symbols. |
| Investor Exchange (IEX) | Historical stock prices are available if traded on IEX. | Requires an API key from IEX Cloud Console. |
| Alpha Vantage | Historical equity data for daily, weekly, and monthly frequencies, 20+ years, and the past 3-5 days of intra-day data. It also has FOREX and sector performance data. | |
| Quandl | Free data sources as listed on their website. | |
| Fama/French | Risk factor portfolio returns. | Used in *Chapter 7, Linear Models – From Risk Factors to Return Forecasts*. |
| TSP Fund Data | Mutual fund prices. | |
| Nasdaq | Latest metadata on traded tickers. | |
| Stooq Index Data | Some equity indices are not available from elsewhere due to licensing issues. | |
| MOEX | Moscow Exchange historical data. | |

The access and retrieval of data follow a similar API for all sources, as illustrated for Yahoo! Finance:

```
import pandas_datareader.data as web
from datetime import datetime
start = '2014'              # accepts strings
end = datetime(2017, 5, 24) # or datetime objects
yahoo= web.DataReader('FB', 'yahoo', start=start, end=end)
yahoo.info()
DatetimeIndex: 856 entries, 2014-01-02 to 2017-05-25
Data columns (total 6 columns):
High         856 non-null float64
Low          856 non-null float64
Open         856 non-null float64
Close        856 non-null float64
Volume       856 non-null int64
```

```
Adj Close    856 non-null float64
dtypes: float64(5), int64(1)
```

## yfinance – scraping data from Yahoo! Finance

`yfinance` aims to provide a reliable and fast way to download historical market data from Yahoo! Finance. The library was originally named `fix-yahoo-finance`. The usage of this library is very straightforward; the notebook `yfinance_demo` illustrates the library's capabilities.

### How to download end-of-day and intraday prices

The `Ticker` object permits the downloading of various data points scraped from Yahoo's website:

```
import yfinance as yf
symbol = 'MSFT'
ticker = yf.Ticker(symbol)
```

The `.history` method obtains historical prices for various periods, from one day to the maximum available, and at different frequencies, whereas intraday is only available for the last several days. To download adjusted OHLCV data at a one-minute frequency and corporate actions, use:

```
data = ticker.history(period='5d',
                      interval='1m',
                      actions=True,
                      auto_adjust=True)
data.info()
DatetimeIndex: 1747 entries, 2019-11-22 09:30:00-05:00 to 2019-11-29 13:00:00-05:00
Data columns (total 7 columns):
Open            1747 non-null float64
High            1747 non-null float64
Low             1747 non-null float64
Close           1747 non-null float64
Volume          1747 non-null int64
Dividends       1747 non-null int64
Stock Splits    1747 non-null int64
```

The notebook also illustrates how to access quarterly and annual financial statements, sustainability scores, analyst recommendations, and upcoming earnings dates.

### How to download the option chain and prices

`yfinance` also provides access to the option expiration dates and prices and other information for various contracts. Using the `ticker` instance from the previous example, we get the expiration dates using:

```
ticker.options
('2019-12-05',  '2019-12-12',  '2019-12-19',..)
```

For any of these dates, we can access the option chain and view details for the various put/call contracts as follows:

```
options = ticker.option_chain('2019-12-05')
options.calls.info()
Data columns (total 14 columns):
contractSymbol       35 non-null object
lastTradeDate        35 non-null datetime64[ns]
strike               35 non-null float64
lastPrice            35 non-null float64
bid                  35 non-null float64
ask                  35 non-null float64
change               35 non-null float64
percentChange        35 non-null float64
volume               34 non-null float64
openInterest         35 non-null int64
impliedVolatility    35 non-null float64
inTheMoney           35 non-null bool
contractSize         35 non-null object
currency             35 non-null object
```

The library also permits the use of proxy servers to prevent rate limiting and facilitates the bulk downloading of multiple tickers. The notebook demonstrates the usage of these features as well.

## Quantopian

Quantopian is an investment firm that offers a research platform to crowd-source trading algorithms. Registration is free, and members can research trading ideas using a broad variety of data sources. It also offers an environment to backtest the algorithm against historical data, as well as to forward-test it out of sample with live data. It awards investment allocations for top-performing algorithms whose authors are entitled to a 10 percent (at the time of writing) profit share.

The Quantopian research platform consists of a Jupyter Notebook environment for research and development for alpha-factor research and performance analysis. There is also an **interactive development environment (IDE)** for coding algorithmic strategies and backtesting the result using historical data since 2002 with minute-bar frequency.

Users can also simulate algorithms with live data, which is known as *paper trading*. Quantopian provides various market datasets, including U.S. equity and futures price and volume data at a one-minute frequency, and U.S. equity corporate fundamentals, and it also integrates numerous alternative datasets.

We will dive into the Quantopian platform in much more detail in *Chapter 4*, *Financial Feature Engineering – How to Research Alpha Factors*, and rely on its functionality throughout the book, so feel free to open an account right away. (Refer to the GitHub repository for more details.)

## Zipline

Zipline is the algorithmic trading library that powers the Quantopian backtesting and live-trading platform. It is also available offline to develop a strategy using a limited number of free data bundles that can be ingested and used to test the performance of trading ideas before porting the result to the online Quantopian platform for paper and live trading.

Zipline requires a custom environment—view the instructions at the beginning of the notebook `zipline_data_demo.ipynb` The following code illustrates how Zipline permits us to access daily stock data for a range of companies. You can run Zipline scripts in the Jupyter Notebook using the magic function of the same name.

First, you need to initialize the context with the desired security symbols. We'll also use a counter variable. Then, Zipline calls `handle_data`, where we use the `data.history()` method to look back a single period and append the data for the last day to a `.csv` file:

```
%load_ext zipline
%%zipline --start 2010-1-1 --end 2018-1-1 --data-frequency daily
from zipline.api import order_target, record, symbol
def initialize(context):
    context.i = 0
    context.assets = [symbol('FB'), symbol('GOOG'), symbol('AMZN')]

def handle_data(context, data):
    df = data.history(context.assets, fields=['price', 'volume'],
                      bar_count=1, frequency="1d")
    df = df.to_frame().reset_index()

    if context.i == 0:
        df.columns = ['date', 'asset', 'price', 'volume']
        df.to_csv('stock_data.csv', index=False)
    else:
        df.to_csv('stock_data.csv', index=False, mode='a', header=None)
    context.i += 1
df = pd.read_csv('stock_data.csv')
df.date = pd.to_datetime(df.date)
df.set_index('date').groupby('asset').price.plot(lw=2, legend=True,
       figsize=(14, 6));
```

We get the following plot for the preceding code:


Figure 2.9: Zipline data access

We will explore the capabilities of Zipline, and especially the online Quantopian platform, in more detail in the coming chapters.

## Quandl

Quandl provides a broad range of data sources, both free and as a subscription, using a Python API. Register and obtain a free API key to make more than 50 calls per day. Quandl data covers multiple asset classes beyond equities and includes FX, fixed income, indexes, futures and options, and commodities.

API usage is straightforward, well-documented, and flexible, with numerous methods beyond single-series downloads, for example, including bulk downloads or metadata searches.

The following call obtains oil prices from 1986 onward, as quoted by the U.S. Department of Energy:

```
import quandl
oil = quandl.get('EIA/PET_RWTC_D').squeeze()
oil.plot(lw=2, title='WTI Crude Oil Price')
```

We get this plot for the preceding code:

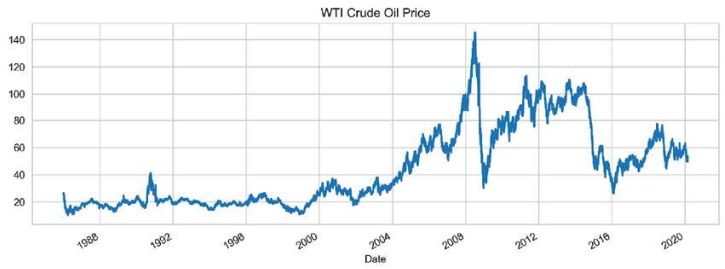


Figure 2.10: Quandl oil price example

## Other market data providers

A broad variety of providers offer market data for various asset classes. Examples in relevant categories include:

- Exchanges derive a growing share of their revenues from an ever-broader range of data services, typically using a subscription.
- Bloomberg and Thomson Reuters have long been the leading data aggregators with a combined share of over 55 percent in the $28.5 billion financial data market. Smaller rivals, such as FactSet, are growing or emerging, such as money.net, Quandl, Trading Economics, and Barchart.
- Specialist data providers abound. One example is LOBSTER, which aggregates Nasdaq order-book data in real time.
- Free data providers include Alpha Vantage, which offers Python APIs for real-time equity, FX, and cryptocurrency market data, as well as technical indicators.

- Crowd-sourced investment firms that provide research platforms with data access include, in addition to Quantopian, Alpha Trading Labs, launched in March 2018, which provides HFT infrastructure and data.

# How to work with fundamental data

Fundamental data pertains to the economic drivers that determine the value of securities. The nature of the data depends on the asset class:

- For equities and corporate credit, it includes corporate financials, as well as industry and economy-wide data.
- For government bonds, it includes international macro data and foreign exchange.
- For commodities, it includes asset-specific supply-and-demand determinants, such as weather data for crops.

We will focus on equity fundamentals for the U.S., where data is easier to access. There are some 13,000+ public companies worldwide that generate 2 million pages of annual reports and more than 30,000 hours of earnings calls. In algorithmic trading, fundamental data and features engineered from this data may be used to derive trading signals directly, for example, as value indicators, and are an essential input for predictive models, including ML models.

## Financial statement data

The Securities and Exchange Commission (SEC) requires U.S. issuers—that is, listed companies and securities, including mutual funds—to file three quarterly financial statements (Form 10-Q) and one annual report (Form 10-K), in addition to various other regulatory filing requirements.

Since the early 1990s, the SEC made these filings available through its **Electronic Data Gathering, Analysis, and Retrieval** (**EDGAR**) system. They constitute the primary data source for the fundamental analysis of equity and other securities, such as corporate credit, where the value depends on the business prospects and financial health of the issuer.

### Automated processing – XBRL

Automated analysis of regulatory filings has become much easier since the SEC introduced **XBRL**, which is a free, open, and global standard for the electronic representation and exchange of business reports. XBRL is based on XML; it relies on taxonomies that define the meaning of the elements of a report and map to tags that highlight the corresponding information in the electronic version of the report. One such taxonomy represents the U.S. **Generally Accepted Accounting Principles** (**GAAP**).

The SEC introduced voluntary XBRL filings in 2005 in response to accounting scandals before requiring this format for all filers as of 2009, and it continues to expand the mandatory coverage to other regulatory filings. The SEC maintains a website that lists the current taxonomies that

shape the content of different filings and can be used to extract specific items.

The following datasets provide information extracted from EX-101 attachments submitted to the commission in a flattened data format to assist users in consuming data for analysis. The data reflects selected information from the XBRL-tagged financial statements. It currently includes numeric data from the quarterly and annual financial statements, as well as certain additional fields, for example, **Standard Industrial Classification** (**SIC**).

There are several avenues to track and access fundamental data reported to the SEC:

- As part of the EDGAR **Public Dissemination Service** (**PDS**), electronic feeds of accepted filings are available for a fee.
- The SEC updates the RSS feeds, which list the structured disclosure submissions, every 10 minutes.
- There are public index files for the retrieval of all filings through FTP for automated processing.
- The financial statement (and notes) datasets contain parsed XBRL data from all financial statements and the accompanying notes.

The SEC also publishes log files containing the internet search traffic for EDGAR filings through SEC.gov, albeit with a six month delay.

### Building a fundamental data time series

The scope of the data in the financial statement and notes datasets consists of numeric data extracted from the primary financial statements (balance sheet, income statement, cash flows, changes in equity, and comprehensive income) and footnotes on those statements. The available data is from as early as 2009.

#### Extracting the financial statements and notes dataset

The following code downloads and extracts all historical filings contained in the **financial statement and notes** (**FSN**) datasets for the given range of quarters (refer to `edgar_xbrl.ipynb` for additional details):

```
SEC_URL = 'https://www.sec.gov/files/dera/data/financial-statement-and-notes-data-sets/'
first_year, this_year, this_quarter = 2014, 2018, 3
past_years = range(2014, this_year)
filing_periods = [(y, q) for y in past_years for q in range(1, 5)]
filing_periods.extend([(this_year, q) for q in range(1, this_quarter +
                                                     1)])
for i, (yr, qtr) in enumerate(filing_periods, 1):
    filing = f'{yr}q{qtr}_notes.zip'
    path = data_path / f'{yr}_{qtr}' / 'source'
    response = requests.get(SEC_URL + filing).content
    with ZipFile(BytesIO(response)) as zip_file:
        for file in zip_file.namelist():
            local_file = path / file
            with local_file.open('wb') as output:
```

```
                for line in zip_file.open(file).readlines():
                    output.write(line)
```

The data is fairly large, and to enable faster access than the original text files permit, it is better to convert the text files into a binary, Parquet columnar format (refer to the *Efficient data storage with pandas* section later in this chapter for a performance comparison of various data-storage options that are compatible with pandas DataFrames):

```
for f in data_path.glob('**/*.tsv'):
    file_name = f.stem  + '.parquet'
    path = Path(f.parents[1]) / 'parquet'
    df = pd.read_csv(f, sep='\t', encoding='latin1', low_memory=False)
    df.to_parquet(path / file_name)
```

For each quarter, the FSN data is organized into eight file sets that contain information about submissions, numbers, taxonomy tags, presentation, and more. Each dataset consists of rows and fields and is provided as a tab-delimited text file:

| File | Dataset | Description |
|---|---|---|
| `SUB` | Submission | Identifies each XBRL submission by company, form, date, and so on |
| `TAG` | Tag | Defines and explains each taxonomy tag |
| `DIM` | Dimension | Adds detail to numeric and plain text data |
| `NUM` | Numeric | One row for each distinct data point in filing |
| `TXT` | Plain text | Contains all non-numeric XBRL fields |
| `REN` | Rendering | Information for rendering on the SEC website |
| `PRE` | Presentation | Details of tag and number presentation in primary statements |
| `CAL` | Calculation | Shows the arithmetic relationships among tags |

**Retrieving all quarterly Apple filings**

The submission dataset contains the unique identifiers required to retrieve the filings: the **Central Index Key** (**CIK**) and the **Accession Number** (**adsh**). The following shows some of the information about Apple's 2018Q1 10-Q filing:

```
apple = sub[sub.name == 'APPLE INC'].T.dropna().squeeze()
key_cols = ['name', 'adsh', 'cik', 'name', 'sic', 'countryba',
```

```
            'stprba', 'cityba', 'zipba', 'bas1', 'form', 'period',
            'fy', 'fp', 'filed']
apple.loc[key_cols]
name                    APPLE INC
adsh                    0000320193-18-000070
cik                     320193
name                    APPLE INC
sic                     3571
countryba               US
stprba                  CA
cityba                  CUPERTINO
zipba                   95014
bas1                    ONE APPLE PARK WAY
form                    10-Q
period                  20180331
fy                      2018
fp                      Q2
filed                   20180502
```

Using the CIK, we can identify all of the historical quarterly filings available for Apple and combine this information to obtain 26 10-Q forms and 9 annual 10-K forms:

```
aapl_subs = pd.DataFrame()
for sub in data_path.glob('**/sub.parquet'):
    sub = pd.read_parquet(sub)
    aapl_sub = sub[(sub.cik.astype(int) == apple.cik) &
                   (sub.form.isin(['10-Q', '10-K']))]
    aapl_subs = pd.concat([aapl_subs, aapl_sub])
aapl_subs.form.value_counts()
10-Q    15
10-K     4
```

With the accession number for each filing, we can now rely on the taxonomies to select the appropriate XBRL tags (listed in the `TAG` file) from the `NUM` and `TXT` files to obtain the numerical or textual/footnote data points of interest.

First, let's extract all of the numerical data that is available from the 19 Apple filings:

```
aapl_nums = pd.DataFrame()
for num in data_path.glob('**/num.parquet'):
    num = pd.read_parquet(num).drop('dimh', axis=1)
    aapl_num = num[num.adsh.isin(aapl_subs.adsh)]
    aapl_nums = pd.concat([aapl_nums, aapl_num])
aapl_nums.ddate = pd.to_datetime(aapl_nums.ddate, format='%Y%m%d')
aapl_nums.shape
(28281, 16)
```

**Building a price/earnings time series**

In total, the 9 years of filing history provide us with over 28,000 numerical values. We can select a useful field, such as **earnings per diluted**

**share (EPS)**, that we can combine with market data to calculate the popular **price-to-earnings (P/E)** valuation ratio.

We do need to take into account, however, that Apple split its stock by 7:1 on June 4, 2014, and adjust the earnings per share values before the split to make the earnings comparable to the price data, which, in its *adjusted* form, accounts for these changes. The following code block shows you how to adjust the earnings data:

```
field = 'EarningsPerShareDiluted'
stock_split = 7
split_date = pd.to_datetime('20140604')
# Filter by tag; keep only values measuring 1 quarter
eps = aapl_nums[(aapl_nums.tag == 'EarningsPerShareDiluted')
                & (aapl_nums.qtrs == 1)].drop('tag', axis=1)
# Keep only most recent data point from each filing
eps = eps.groupby('adsh').apply(lambda x: x.nlargest(n=1, columns=['ddate']))
# Adjust earnings prior to stock split downward
eps.loc[eps.ddate < split_date,'value'] = eps.loc[eps.ddate <
        split_date, 'value'].div(7)
eps = eps[['ddate', 'value']].set_index('ddate').squeeze()
# create trailing 12-months eps from quarterly data
eps = eps.rolling(4, min_periods=4).sum().dropna()
```

We can use Quandl to obtain Apple stock price data since 2009:

```
import pandas_datareader.data as web
symbol = 'AAPL.US'
aapl_stock = web.DataReader(symbol, 'quandl', start=eps.index.min())
aapl_stock = aapl_stock.resample('D').last() # ensure dates align with
                                             eps data
```

Now we have the data to compute the trailing 12-month P/E ratio for the entire period:

```
pe = aapl_stock.AdjClose.to_frame('price').join(eps.to_frame('eps'))
pe = pe.fillna(method='ffill').dropna()
pe['P/E Ratio'] = pe.price.div(pe.eps)
axes = pe.plot(subplots=True, figsize=(16,8), legend=False, lw=2);
```

We get the following plot from the preceding code:

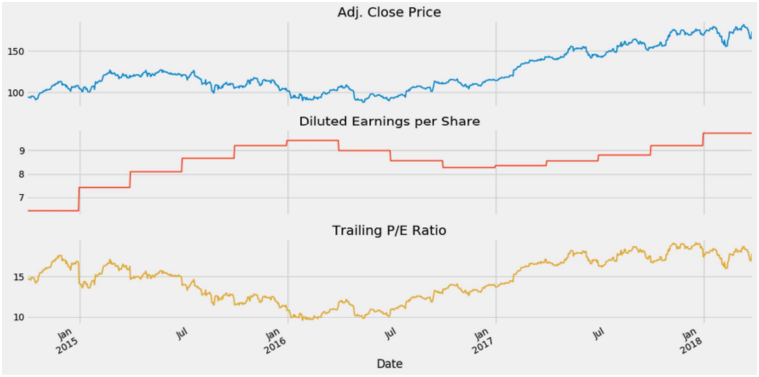

Figure 2.11: Trailing P/E ratio from EDGAR filings

### Other fundamental data sources

There are numerous other sources for fundamental data. Many are accessible using the `pandas_datareader` module that was introduced earlier. Additional data is available from certain organizations directly, such as the IMF, the World Bank, or major national statistical agencies around the world (refer to the *references* section on GitHub).

#### pandas-datareader – macro and industry data

The `pandas-datareader` library facilitates access according to the conventions introduced at the end of the preceding section on market data. It covers APIs for numerous global fundamental macro- and industry-data sources, including the following:

- Kenneth French's data library: Market data on portfolios capturing returns on key risk factors like size, value, and momentum factors, disaggregated by industry (refer to *Chapter 4*, *Financial Feature Engineering – How to Research Alpha Factors*)
- St. Louis FED (FRED): Federal Reserve data on the U.S. economy and financial markets
- World Bank: Global database on long-term, lower-frequency economic and social development and demographics
- OECD: Similar to the World Bank data for OECD countries
- Enigma: Various datasets, including alternative sources
- Eurostat: EU-focused economic, social, and demographic data

## Efficient data storage with pandas

We'll be using many different datasets in this book, and it's worth comparing the main formats for efficiency and performance. In particular, we'll compare the following:

- **CSV**: Comma-separated, standard flat text file format.
- **HDF5**: Hierarchical data format, developed initially at the National Center for Supercomputing Applications. It is a fast and scalable storage format for numerical data, available in pandas using the PyTables library.
- **Parquet**: Part of the Apache Hadoop ecosystem, a binary, columnar storage format that provides efficient data compression and encoding and has been developed by Cloudera and Twitter. It is available for pandas through the pyarrow library, led by Wes McKinney, the original author of pandas.

The `storage_benchmark.ipynb` notebook compares the performance of the preceding libraries using a test DataFrame that can be configured to contain numerical or text data, or both. For the HDF5 library, we test both the fixed and table formats. The table format allows for queries and can be appended to.

The following charts illustrate the read and write performance for 100,000 rows with either 1,000 columns of random floats and 1,000 columns of a random 10-character string, or just 2,000 float columns (on a log scale):

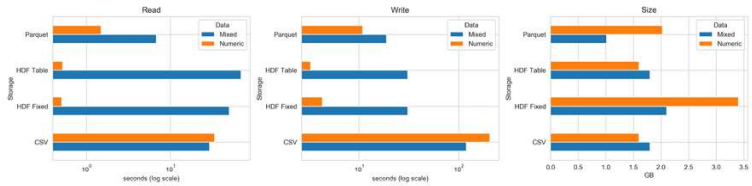


Figure 2.12: Storage benchmarks

The left panel shows that, for purely numerical data, the HDF5 format performs best by far, and the table format also shares with CSV the smallest memory footprint at 1.6 GB. The `fixed` format uses twice as much space, while the `parquet` format uses 2 GB.

For a mix of numerical and text data, Parquet is the best choice for read and write operations. HDF5 has an advantage with *read* in relation to CSV, but it is slower with *write* because it pickles text data.

The notebook illustrates how to configure, test, and collect the timing using the `%%timeit` cell magic and, at the same time, demonstrates the usage of the related pandas commands that are required to use these storage formats.

# Summary

This chapter introduced the market and fundamental data sources that form the backbone of most trading strategies. You learned about the various ways to access this data and how to preprocess the raw information so that you can begin extracting trading signals using the ML techniques that we will be introducing shortly.

In the next chapter, before moving on to the design and evaluation of trading strategies and the use of ML models, we need to cover alternative datasets that have emerged in recent years and have been a significant driver of the popularity of ML for algorithmic trading.