2
Market and Fundamental Data

Data has always been an essential driver of trading, and traders have long made efforts to gain an advantage by having access to superior information. These efforts date back at least to the rumors that the House Rothschild benefited handsomely from bond purchases upon advance news about the British victory at Waterloo 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 exchange in New York has dropped to close to the theoretical limit of eight milliseconds as traders compete to exploit arbitrage opportunities.

Traditionally, investment strategies mostly relied on publicly available data, with limited efforts to create or acquire private datasets. In the case of equities, fundamental strategies used financial models built on reported financials, possibly combined with industry or macro data. Strategies motivated by technical analysis extract signals from market data, such as prices and volumes.

Machine learning (**ML**) algorithms can exploit market and fundamental data more efficiently, in particular when combined with alternative data, which is the topic of the next chapter. We will address several techniques that focus on market and fundamental data in later chapters, such as classic and modern time-series techniques, including **recurrent neural networks** (**RNNs**).

This chapter introduces market and fundamental data sources and the environment in which they are created. Familiarity with various types of orders and the trading infrastructure matters because they affect backtest simulations of a trading strategy. We also illustrate how to use Python to access and work with trading and financial statement data.

In particular, this chapter will cover the following topics:

- How market microstructure shapes market data
- How to reconstruct the order book from tick data using Nasdaq ITCH

- How to summarize tick data using various types of bars
- How to work with **eXtensible Business Reporting Language** (**XBRL**)-encoded electronic filings
- How to parse and combine market and fundamental data to create a P/E series
- How to access various market and fundamental data sources using Python

How to work with market data

Market data results from the placement and processing of buy and sell orders in the course of the trading of financial instruments on the many marketplaces. The data reflects the institutional environment of trading venues, including the rules and regulations that govern orders, trade execution, and price formation.

Algorithmic traders use ML algorithms to analyze the flow of buy and sell orders and the resulting volume and price statistics to extract trade signals or features 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. Then, we will take a look at how tick data can be reconstructed from the order book source. Next, we will highlight several methods that regularize tick data and aim to maximize the information content. Finally, we will illustrate how to access various market data provider interfaces and highlight several providers.

Market microstructure

Market microstructure is the branch of financial economics that investigates the trading process and the organization of related markets. The institutional details are quite complex and diverse across asset classes and their derivatives, trading venues, and geographies. We will only give a brief overview of key concepts before we dive into the data generated by trading. The references on GitHub link to several sources that treat this subject in great detail.

Marketplaces

Trading in financial instruments occurs in organized, mostly electronic exchanges, and over the counter. An exchange is a central marketplace where buyers and sellers meet, and buyers compete with each other for the highest bid while sellers compete for the lowest offer.

There are many exchanges and alternative trading venues across the US and abroad. The following table lists some of the larger global exchanges and the trading volumes for the 12 months concluded 03/2018 in various asset classes, including derivatives. Typically, a minority of financial instruments accounts for most trading:

Exchange	Stocks				
	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

Exchanges may rely on bilateral trading or order-driven systems, where buy and sell orders are matched according to certain rules. 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.

Many exchanges use intermediaries that provide liquidity, that is, the ability to trade, by making markets in certain securities. The NYSE, for example, usually has a single designated market maker who ensures orderly trading for each security, while the **National Association of Securities Dealers Automated Quotations** (**Nasdaq**) has several. Intermediaries can act as dealers that trade as principals on their own behalf, or brokers that trade as agents on behalf of others.

Exchanges used to be member-owned but have often moved to corporate ownership as market reforms have increased competition. The NYSE dates back to 1792, whereas the Nasdaq started 1971 as the world's first electronic stock market and took over most stock trades that had been executed OTC. In US equity markets alone, trading is fragmented across 13 exchanges and over 40 alternative trading venues, each reporting trades to the consolidated tape, but at different latencies.

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

Numerous other conditions can be attached to orders—all or none orders prevent partial execution and are only filled if a specified number of shares is available, and can be valid for the 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.

Working with order book data

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

The trading activity is reflected in numerous messages about trade orders sent by market participants. These messages typically conform to the electronic **Financial Information eXchange** (**FIX**) communications protocol for real-time exchange of securities transactions and market data or a native exchange protocol.

The FIX protocol

Just like SWIFT is the message protocol for back-office (example, for trade-settlement) messaging, the FIX protocol is the de facto messaging standard for communication before and during, trade execution between exchanges, banks, brokers, clearing firms, and other market participants. Fidelity Investments and Salomon Brothers introduced FIX in 1992 to facilitate electronic communication between broker-dealers and institutional clients who by 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_string|98=2|108=34|141=Y|553=12345|554=passw0rd!|1
0=131|
```

There are a few open source FIX implementations in python that can be used to formulate and parse FIX messages. Interactive Brokers offer a FIX-based **computer-to-computer interface** (**CTCI**) for automated trading (see the resources section for this chapter in the GitHub repo).

Nasdaq TotalView-ITCH Order Book data

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

As a result, it allows for the reconstruction of the order book that keeps track of the list of active-limit buy and sell orders for a specific security or financial instrument. 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 it is 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, the 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 the Nasdaq opening and closing crosses and Nasdaq IPO/Halt cross.

Parsing binary ITCH 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.

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

The following table shows the frequency of the most common message types for the sample file date March 29, 2018:

Message type	Order book impact	Number of messages
`A`	New unattributed limit order	136,522,761
`D`	Order canceled	133,811,007
`U`	Order canceled and replaced	21,941,015
`E`	Full or partial execution; possibly multiple messages for the same original order	6,687,379

X	Modified after partial cancellation	5,088,959
F	Add attributed order	2,718,602
P	Trade Message (non-cross)	1,120,861
C	Executed in whole or in part at a price different from the initial display price	157,442
Q	Cross Trade Message	17,233

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	F	Add Order MPID attribution message
Stock locate	1	2	Integer	Locate code identifying the security
Tracking number	3	2	Integer	Nasdaq internal tracking number
Timestamp	5	6	Integer	Nanoseconds since midnight
Order reference number	11	8	Integer	Unique reference number of the new order
Buy/sell indicator	19	1	Alpha	The type of order: B = Buy Order, S = Sell Order
Shares	20	4	Integer	Number of shares for 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
Attribution	36	4	Alpha	Nasdaq Market participant identifier associated with the order

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

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

1. The ITCH parser relies on the message specifications provided as a `.csv` file (created by `create_message_spec.py`) and 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 `namedtuples` 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 post-processing as defined in the `format_alpha` function:

```
def format_alpha(mtype, data):
    for col in alpha_formats.get(mtype).keys():
        if mtype != 'R' and col == 'stock': # stock name only in
                                            summary message 'R'
            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)) # int encoding
    return data
```

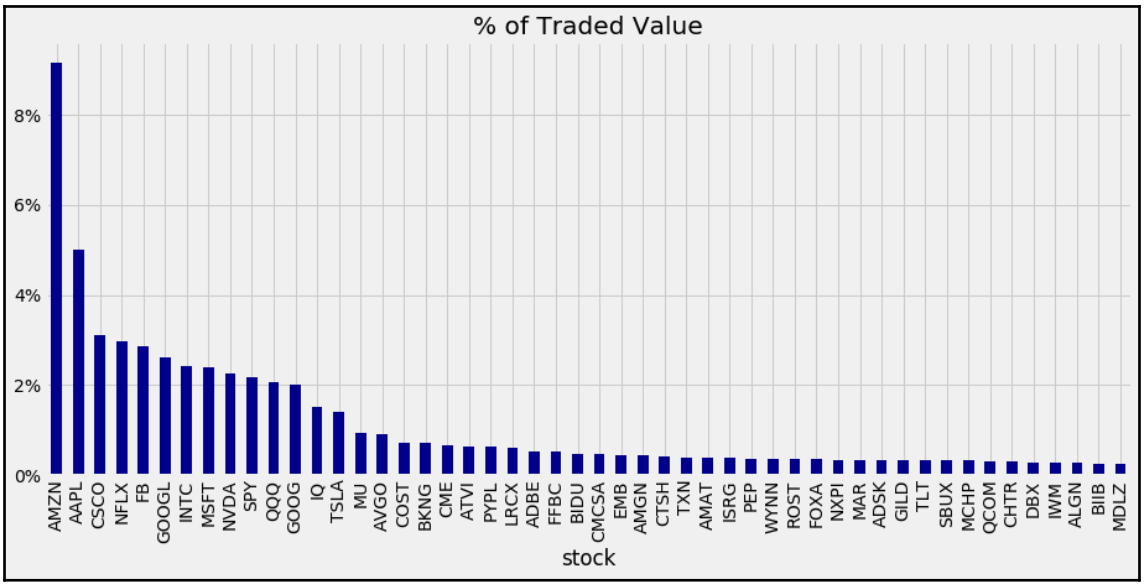
4. The binary file for a single day contains over 300,000,000 messages worth over 9 GB. The script appends the parsed result iteratively to a file in the fast `HDF5` format to avoid memory constraints (see last section in this chapter for more on this format). The following (simplified) code processes 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
```

5. As expected, a small number of the over 8,500 equity securities traded on this day account for most trades:

```
with pd.HDFStore(hdf_store) as store:
    stocks = store['R'].loc[:, ['stock_locate', 'stock']]
    trades = store['P'].append(store['Q'].rename(columns=
                         {'cross_price': 'price'}).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='% of Traded Value')
plt.gca().yaxis.set_major_formatter(FuncFormatter(lambda y, _:
                                    '{:.0%}'.format(y)))
```

We get the following plot for the graph:



Reconstructing 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, but we are ignoring this aspect here.

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, slightly simplified, see notebook):

```
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 are 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. To reconstruct the order book for a given level of depth requires the following steps:

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

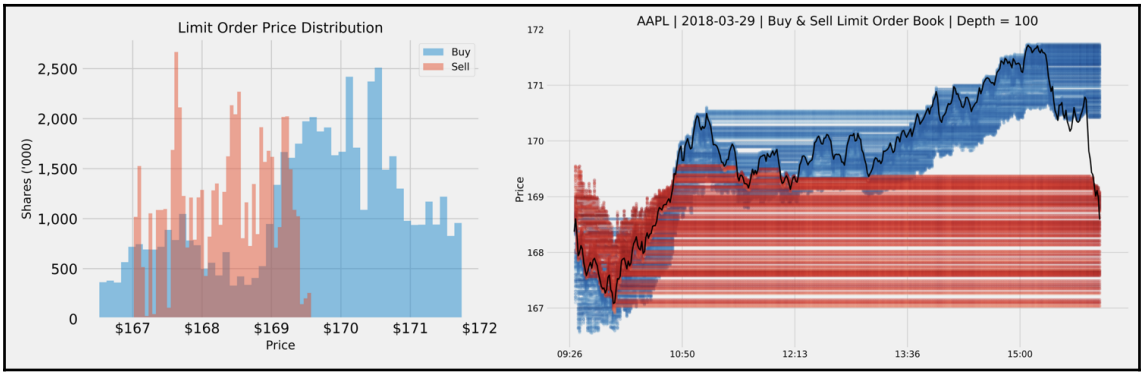
```
def add_orders(orders, buysell, nlevels):
    new_order = []
    items = sorted(orders.copy().items())
    if buysell == -1:
        items = reversed(items)
    for i, (p, s) in enumerate(items, 1):
        new_order.append((p, s))
        if i == nlevels:
            break
    return orders, new_order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

2. 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
```

The number of orders at different price levels, highlighted in the following screenshot using different intensities for buy and sell orders, visualizes the depth of liquidity at any given point in time. 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 (see notebook for details):



Regularizing tick data

The trade data is indexed by nanoseconds 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 improve the statistical properties, we need to resample and regularize the tick data by aggregating the trading activity.

We typically collect the open (first), low, high, and closing (last) price for the aggregated period, alongside the **volume-weighted average price** (**VWAP**), the number of shares traded, and the timestamp associated with the data.

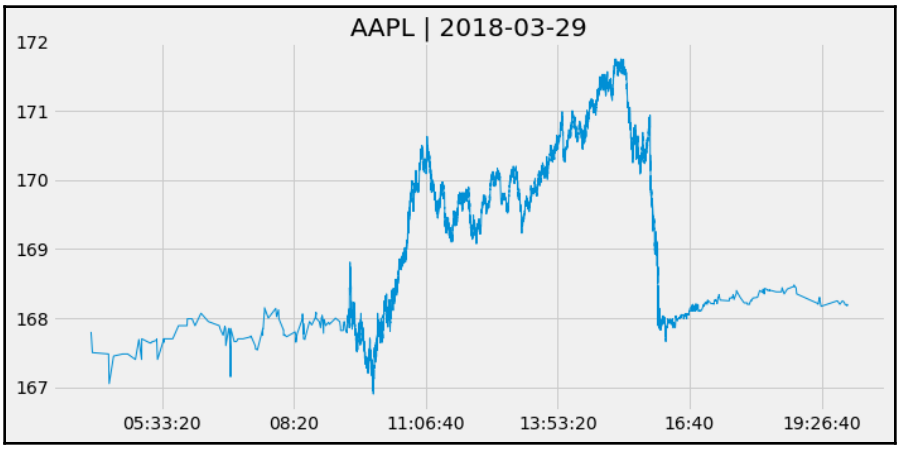
See the `normalize_tick_data.ipynb` notebook in the folder for this chapter on GitHub for additional detail.

Tick bars

A plot of the raw tick price and volume data for `AAPL` looks as follows:

```
stock, date = 'AAPL', '20180329'
title = '{} | {}'.format(stock, pd.to_datetime(date).date()

with pd.HDFStore(itch_store) as store:
    s = store['S'].set_index('event_code') # system events
    s.timestamp = s.timestamp.add(pd.to_datetime(date)).dt.time
    market_open = s.loc['Q', 'timestamp']
    market_close = s.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)

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)
```

We get the following plot for the preceding code:



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)
```

Time bars

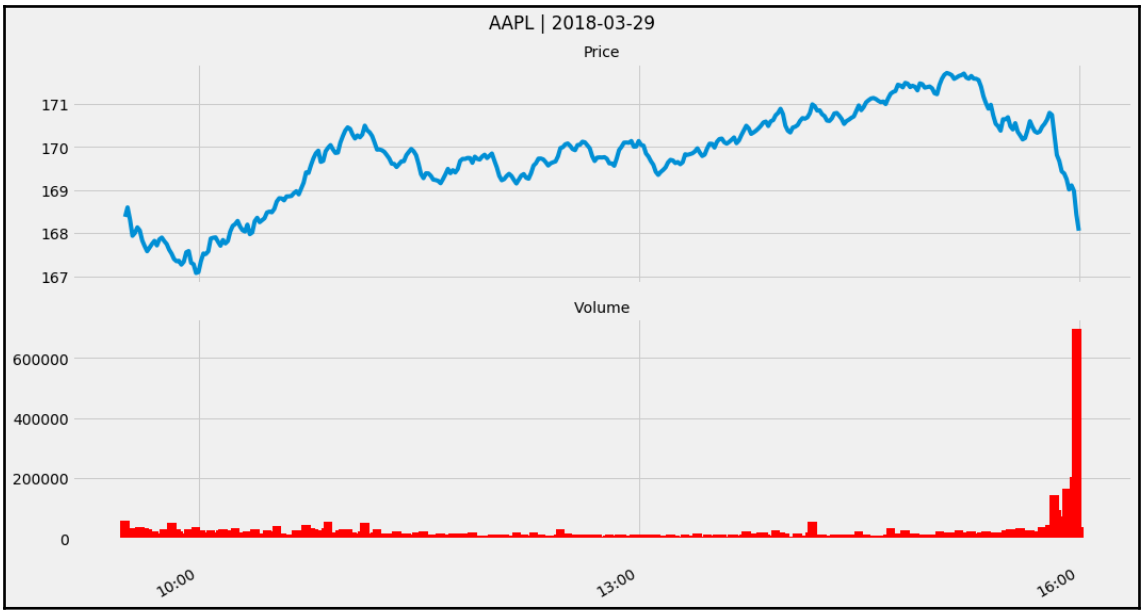
Time bars involve trade aggregation by period:

```
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.resample('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):
    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)
```

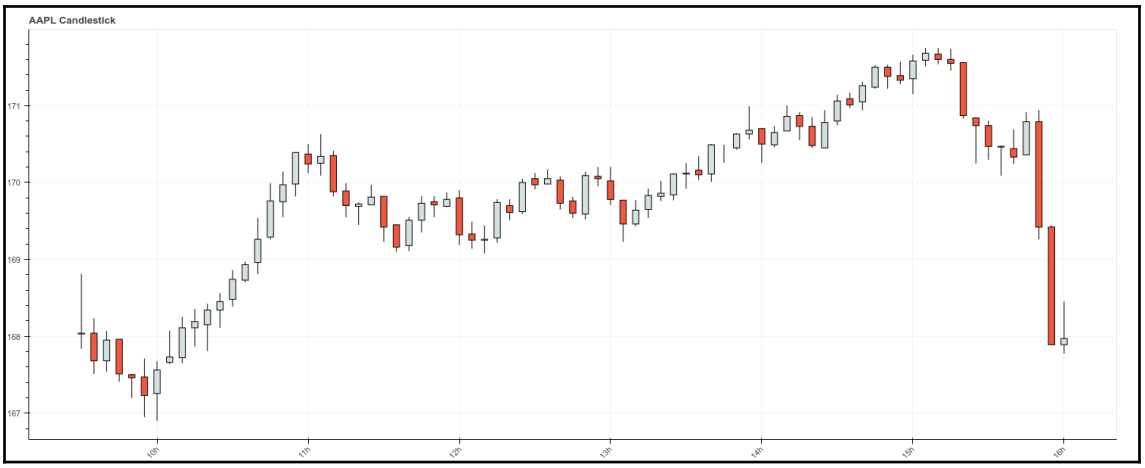
We get the following plot for the preceding code:



Or as a candlestick chart using the `bokeh` plotting library:

```
resampled = trades.resample('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)
```

Take a look at the following screenshot:



Plotting AAPL Candlestick

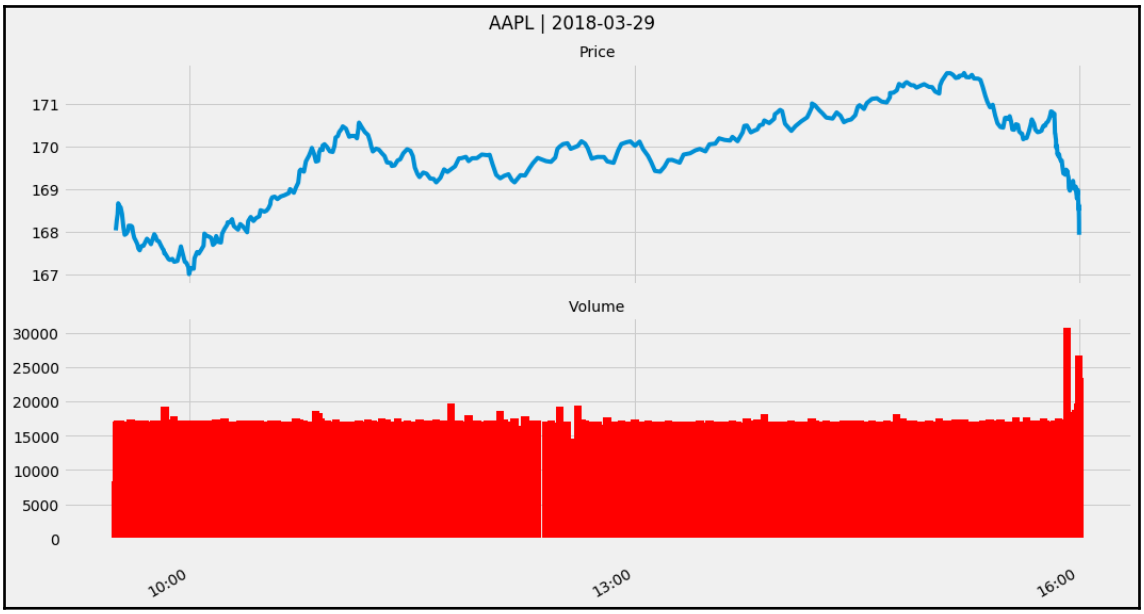
Volume bars

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, and 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:

```
trades_per_min = trades.shares.sum()/(60*7.5) # 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 following plot for the preceding code:



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 price to produce dollar bars.

API access to market data

There are several options to access market data via API using Python. We first present a few sources built into the `pandas` library. Then we briefly introduce the trading platform Quantopian, the data provider Quandl and the backtesting library that we will use later in the book, and list several additional options to access various types of market data. The folder 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

The download of the content of one or more `html` tables works as follows, for instance for the constituents of the `S&P500` index from Wikipedia:

```
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):
Ticker 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
Location                  505 non-null object
Date first added[3][4]    398 non-null object
CIK                       505 non-null int64
Founded                   139 non-null object
```

pandas-datareader for market data

pandas used to facilitate access to data providers' APIs directly, but this functionality has moved to the related `pandas-datareader` library. The stability of the APIs varies with provider policies, and as of June 2o18 at version 0.7, the following sources are available:

Source	**Scope**	**Comment**
Yahoo! Finance	EOD price, dividends, split data for stock and FX pairs	Unstable
Tiingo	EOD prices on equities, mutual funds, and ETFs	Free registration required
The **Investors Exchange** (**IEX**)	Historical stock prices, order-book data	Limited to five years
Robinhood	EOD equity prices	Limited to one year
Quandl	Marketplace for a broad range of asset prices	Premium data require a subscription

Nasdaq	Latest ticker symbols traded on Nasdaq with some additional info	
Stooq	Some stock market index data	
MOEX	Moscow Stock Exchange Data	
Alpha Vantage	EOD stock prices and FX pairs	
Fama/French	Factor returns and research portfolios from the FF Data Library	

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)
```

The Investor Exchange

IEX is an alternative exchange started in response to the HFT controversy and portrayed in Michael Lewis' controversial Flash Boys. It aims to slow down the speed of trading to create a more level playing field and has been growing rapidly since launch in 2016 while still small with a market share of around 2.5% in June 2018.

In addition to historical EOD price and volume data, IEX provides real-time depth of book quotations that offer an aggregated size of orders by price and side. This service also includes last trade price and size information:

```
book = web.get_iex_book('AAPL')
orders = pd.concat([pd.DataFrame(book[side]).assign(side=side) for side in
['bids', 'asks']])
orders.sort_values('timestamp').head()
```

```
  price  size timestamp      side
4 140.00  100  1528983003604 bids
3 175.30  100  1528983900163 bids
3 205.80  100  1528983900163 asks
1 187.00  200  1528996876005 bids
2 186.29  100  1528997296755 bids
```

See additional examples in the `datareader.ipynb` notebook.

Quantopian

Quantopian is an investment firm that offers a research platform to crowd-source trading algorithms. Upon free registration, it enables members to research trading ideas using a broad variety of data sources. It also offers an environment to backtest the algorithm against historical data, as well forward-test it out-of-sample with live data. It awards investment allocations for top-performing algorithms whose authors are entitled to a 10% (at 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 US equity and futures price and volume data at a one-minute frequency, as well as US equity corporate fundamentals, and integrates numerous alternative datasets.

We will dive into the Quantopian platform in much more detail in `Chapter 4`, *Alpha Factor Research* and rely on its functionality throughout the book, so feel free to open an account right away (see GitHub repo 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.

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', 'volumne']
 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:

We will explore the capabilities of `zipline`, and, in particular, the online Quantopian platform, in more detail in the coming chapters.

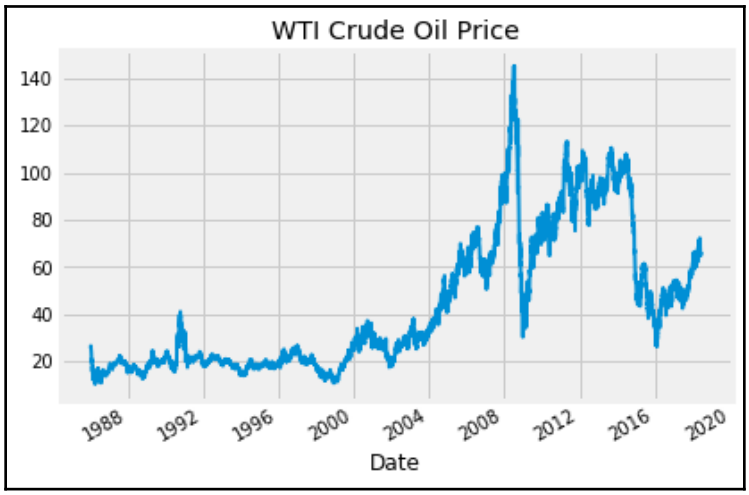
Quandl

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/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 the oil prices since 1986 as quoted by the US 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:



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 subscription.
- Bloomberg and Thomson Reuters have long been the leading data aggregators with a combined share of over 55% in the $28.5 billion financial data market. Smaller rivals, such as FactSet, are growing, or emerging, such as money.net and Quandl as well as Trading Economics or Barchart.
- Specialist data providers abound. One example is LOBSTER, which aggregates Nasdaq order-book data in real-time.
- Free data providers include Alpha Vantage that offers Python APIs for real-time equity, FX, and crypto-currency market data, as well as technical indicators.
- Crowd-sourced investment firms that provide research platforms with data access include, in addition to Quantopian, the Alpha Trading Labs, launched in March 2018, which provide 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 US, where data is easier to access. There are some 13,000+ public companies worldwide that generate 2 million pages of annual reports and 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 machine learning models.

Financial statement data

The **Securities and Exchange Commission** (**SEC**) requires US 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 the XBRL, 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 US **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 since 2009 and 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 the 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 **RSS** feeds every 10 minutes, which list structured disclosure submissions.
- 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 data is available 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 (see `edgar_xbrl.ipynb` for addition details):

```
SEC_URL =
'https://www.sec.gov/files/dera/data/financial-statement-and-notes-data-set
s/'
```

```
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 to binary, columnar parquet format (see *Efficient data storage with pandas* section in this chapter for a performance comparison of various data-storage options 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 SEC website
`PRE`	Presentation	Detail on the tag and number presentation in primary statements
`CAL`	Calculation	Shows 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 central index key, we can identify all historical quarterly filings available for `Apple`, and combine this information to obtain 26 Forms `10-Q` and nine annual Forms `10-K`:

```
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 numerical data available from the 19 Apple filings:

```
aapl_nums = pd.DataFrame()
for num in data_path.glob('**/num.parquet'):
    num = pd.read_parquet(num)
    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 nine 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/Earnings** (**P/E**) valuation ratio.

We do need to take into account, however, that Apple split its stock 7:1 on June 4, 2014, and Adjusted Earnings per Share before the split to make earnings comparable, as illustrated in the following code block:

```
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()
eps = eps.rolling(4, min_periods=4).sum().dropna() # create trailing
                  12-months eps from quarterly data
```

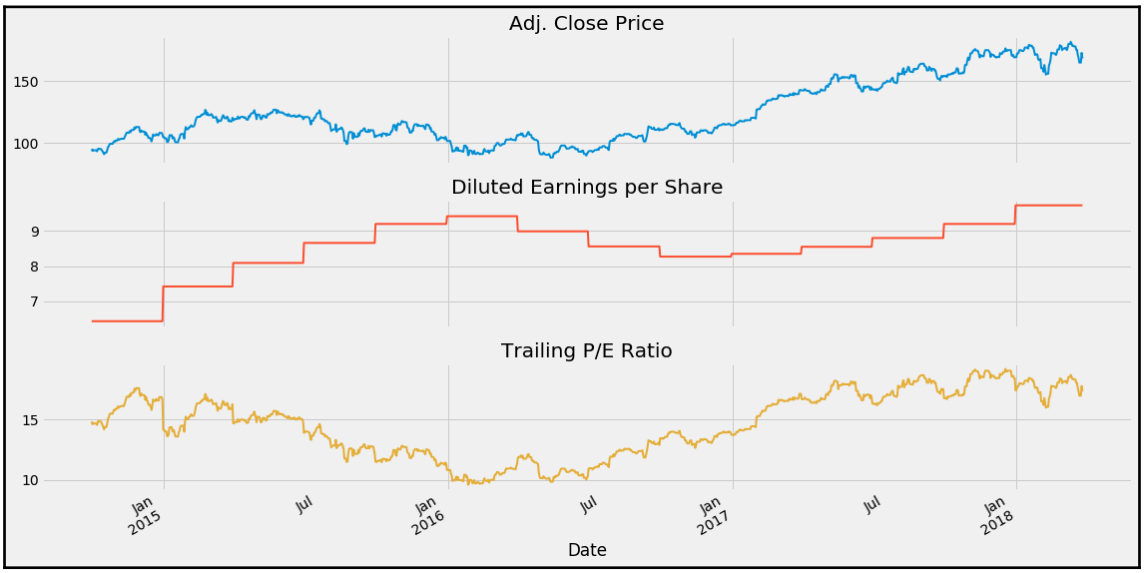
We can use Quandl to obtain Apple stock price data since 2009:

```
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 for the preceding code:



Other fundamental data sources

There are numerous other sources for fundamental data. Many are accessible using the `pandas_datareader` module introduced earlier. Additional data is available from certain organizations directly, such as the IMF, World Bank, or major national statistical agencies around the world (see references 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 size, value, and momentum factors, disaggregated industry
- **St.Louis FED (FRED)**: Federal Reserve data on the US economy and financial markets
- **World Bank**: Global database on long-term, lower-frequency economic and social development and demographics
- **OECD**: Similar for OECD countries
- **Enigma**: Various datasets, including alternative sources
- **Eurostat**: EU-focused economics, social and demographic data

Efficient data storage with pandas

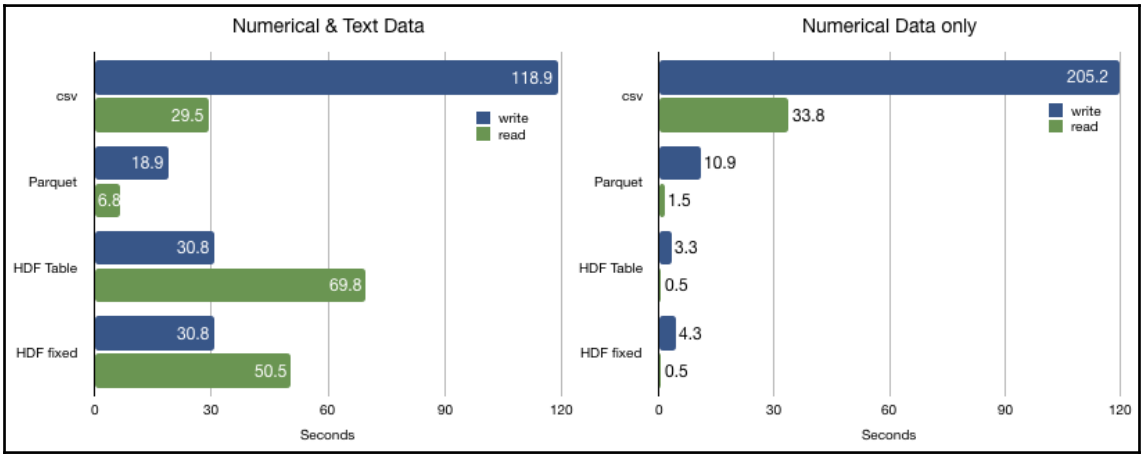
We'll be using many different data sets in this book, and it's worth comparing the main formats for efficiency and performance. In particular, we compare the following:

- **CSV**: Comma-separated, standard flat text file format.
- **HDF5**: Hierarchical data format, developed initially at the National Center for Supercomputing, is a fast and scalable storage format for numerical data, available in pandas using the `PyTables` library.
- **Parquet**: A binary, columnar storage format, part of the Apache Hadoop ecosystem, 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` format. 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:

- For purely numerical data, the HDF5 format performs best, and the table format also shares with CSV the smallest memory footprint at 1.6 GB. The fixed format uses twice as much space, and the parquet format uses 2 GB.
- For a mix of numerical and text data, `parquet` is significantly faster, and HDF5 uses its advantage on reading relative to CSV (which has very low write performance in both cases):



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 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 numerous ways to access this data, and how to preprocess the raw information so that you can begin extracting trading signals using the machine learning techniques that we will be introducing shortly.

Before we move onto 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.