2

Analyze and Transform Financial Market Data with pandas

The pandas library was invented by Wes McKinney while at the investment management firm **AQR Capital Management**, where he researched macro and credit trading strategies. He built pandas to provide flexible, easy-to-use data structures for data analysis. Since it was open sourced in 2009, pandas has become the standard tool to analyze and transform data using Python.

pandas is well-suited for working with tabular data, like that stored in spreadsheets or databases, and it integrates well with many other data analysis libraries in the Python ecosystem. Its capabilities extend to handling missing data, reshaping datasets, and merging and joining datasets, and it also provides robust tools for loading data from flat files, Excel files, databases, and HDF5 file formats. It's widely used in academia, finance, and many areas of business due to its rich features and ease of use.

This chapter will begin by covering recipes to help you build pandas data structures, primarily focusing on DataFrames and Series, the two primary data structures of pandas. You will learn how these structures allow for a wide variety of operations, such as slicing, indexing, and subsetting large datasets, all of which are crucial in algorithmic trading. From there, you will learn how to inspect and select data from DataFrames.

After you have a firm understanding of manipulating data using pandas, we'll cover recipes for analyses that are common in algorithmic trading, including how to compute asset returns and the volatility of a return se-

ries. The chapter will teach you how to generate a cumulative return series and resample data to different time frames, providing you with the flexibility to analyze data at various granularities. You will also learn how to handle missing data, a common issue in real-world datasets.

Lastly, this chapter will show you how to apply custom functions to time series data. Throughout this chapter, you will see how pandas integrates with other libraries in the scientific Python ecosystem, such as Matplotlib for data visualization, NumPy for numerical operations, and Scikit-Learn for machine learning.

In this chapter, we'll cover the following recipes:

- Diving into pandas index types
- Building pandas Series and DataFrames
- Manipulating and transforming DataFrames
- Examining and selecting data from DataFrames
- Calculating asset returns using pandas
- Measuring the volatility of a return series
- Generating a cumulative return series
- Resampling data for different time frames
- Addressing missing data issues
- Applying custom functions to analyze time series data

Diving into pandas index types

The Index is an immutable sequence that's used for indexing and alignment that serves as the label or key for rows in the DataFrame or elements in a series. It allows for fast lookup and relational operations and as of pandas version 2, it can contain values of any type, including integers, strings, and even tuples. Indexes in pandas are immutable, which makes them safe to share across multiple DataFrames or Series. They also have several built-in methods for common operations, such as sorting, grouping, and set operations such as union and intersection. pandas supports multiple indexes, allowing for complex, hierarchical data organization. This feature is particularly useful when dealing with high-dimen-

sional data such as option chains. We'll see examples of MultiIndexes later in this chapter.

There are seven types of pandas indexes. The differences are dependent on the type of data used to create the index. For example, an Int64Index is an index that's made up of 64-bit integers. pandas is smart enough to create the right type of index, depending on the data used to instantiate it.

How to do it...

We'll start by creating a simple index with a series of integers:

1. Import pandas as the common alias, **`pd`**:

```
import pandas as pd
```

2. Instantiate the **`Index`** class:

```
idx_1 = pd.Index([0, 1, 2, 3, 4, 5, 6, 7, 8, 9])
```

3. Inspect the index:

```
print(idx_1)
```

Running the preceding code generates the index and prints out its type:

```
Index([0, 1, 2, 3, 4, 5, 6, 7, 8, 9], dtype='int64')
```

Figure 2.1: A pandas Int64Index index with 10 values

How it works...

This is the simplest example of creating a pandas index. You can instantiate the **`Index`** class with a one-dimensional, array-like data structure containing the values you want in the index.

There's more...

pandas has several **`Index`** types to support many use cases, including those related to time series analysis. We'll cover examples of the most often-used index types.

DatetimeIndex

A more interesting index type is **`DatetimeIndex`**. This type of index is extremely useful when dealing with time series data:

```
days = pd.date_range("2016-01-01", periods=6, freq="D")
```

This creates an index with six incremental datetime objects starting from 2016-01-01.

You can use many different frequencies, including **`seconds`**:

```
seconds = pd.date_range("2016-01-01", periods=100, freq="s")
```

By default, DatetimeIndexes are "timezone naive." This means they have no time zone information attached. To "localize" **`DatetimeIndex`**, use **`tz_localize`**:

```
seconds_utc = seconds.tz_localize("UTC")
```

Localizing **`DatetimeIndex`** just appends time zone information to the object. It does not adjust the time from one time zone to another. To do that, you can use **`tz_convert`**:

```
seconds_utc.tz_convert("US/Eastern")
```

PeriodIndex

It's possible to create ranges of periods, such as quarters, using the pandas **`period_range`** method:

```
prng = pd.period_range("1990Q1", "2000Q4", freq="Q-NOV")
```

`period_range` returns a **`PeriodIndex`** object that has special labels that represent periods. In this case, the period starts in 1990Q1 and extends through to 2000Q4 at quarterly increments, with the year ending in November.

MultiIndex

`MultiIndex`, also known as a hierarchical index, is a data structure that allows for complex data organization within pandas DataFrames and Series. It allows data spanning multiple dimensions to be represented, even within the inherently two-dimensional structure of a DataFrame, by allowing multiple levels of indices. We'll look at **`MultiIndex`** in detail later in this book, so don't worry if it's not exactly clear how they work yet.

To create a **`MultiIndex`** object, pass a list of tuples to the **`from_tuples`** method:

```
tuples = [
    (pd.Timestamp("2023-07-10"), "WMT"),
    (pd.Timestamp("2023-07-10"), "JPM"),
    (pd.Timestamp("2023-07-10"), "TGT"),
    (pd.Timestamp("2023-07-11"), "WMT"),
    (pd.Timestamp("2023-07-11"), "JPM"),
    (pd.Timestamp("2023-07-11"), "TGT"),
]
midx = pd.MultiIndex.from_tuples(
    tuples,
    names=("date", "symbol")
)
```

There are several ways to create a **`MultiIndex`** object. In this example, we use a list of tuples. Each tuple contains a pandas timestamp and a ticker symbol. Using the list of tuples, you can call the **`from_tuples`** method to create the **`MultiIndex`** object. The result is a two-dimensional index that supports hierarchical DataFrames.

> *IMPORTANT NOTE*

> *Giving the* **`names`** *argument to the* **`from_tuples`** *method is done to give names to the index columns for label-based indexing. This is optional.*

See also

The pandas documentation is very thorough. To learn more about the index types we covered in this recipe, take a look at the following resources:

- Documentation on pandas indexes: https://pandas.pydata.org/docs/reference/api/pandas.Index.html
- Documentation on pandas DatatimeIndexes: https://pandas.pydata.org/docs/reference/api/pandas.DatetimeIndex.html
- Documentation on pandas MultiIndexes: https://pandas.pydata.org/docs/reference/api/pandas.MultiIndex.html

Building pandas Series and DataFrames

A Series is a one-dimensional labeled array that can hold any data type, including integers, floats, strings, and objects. The axis labels of a Series are collectively referred to as the index, which allows for easy data manipulation and access. A key feature of the pandas Series is its ability to handle missing data, represented as a NumPy **`nan`** (Not a Number).

> *IMPORTANT*
>
> *NumPy's* **`nan`** *is a special floating-point value. It is commonly used as a marker for missing data in numerical datasets. The* **`nan`** *value being a float is useful because it can be used in numerical computations and included in arrays of numbers without changing their data type, which aids in maintaining consistent data types in numeric datasets. Unlike other values,* **`nan`** *doesn't equal anything, which is why we need to use functions such as* **`numpy.isnan()`** *to check for* **`nan`**.

Furthermore, the **`Series`** object provides a host of methods for operations such as statistical functions, string manipulation, and even plotting.

A DataFrame is a two-dimensional data structure that can be thought of as a spreadsheet with rows and columns. It's essentially a table of data with rows and columns. The data is aligned in a tabular fashion in rows and columns, and it can be of different types (for example, numbers and strings). DataFrames make manipulating data easy, allowing for opera-

tions such as aggregation, slicing, indexing, subsetting, merging, and reshaping. They also handle missing data gracefully and provide convenient methods for filtering out, filling in, or otherwise manipulating null values.

Getting ready

Before building a Series and DataFrame, make sure you completed the previous recipe and have the indexes stored in memory.

How to do it...

Execute the following steps to construct a DataFrame out of several Series:

1. Import NumPy to create an array of normally distributed random numbers:

```
import numpy as np
```

2. Create a function that returns a NumPy array of randomly distributed numbers with the same length as the **`DatetimeIndex`** object you created earlier:

```
def rnd():
    return np.random.randn(100,)
```

   The return value is a NumPy array of length **`100`**. It's filled with random samples from a normal distribution. Creating a function allows you to return a different set of values each time the function is called.

> *TIP*
>
> *When working with pandas, it's common to use random numbers to populate Series and DataFrames for testing.*

3. Create three pandas Series that you'll use to create the DataFrame:

```
s_1 = pd.Series(rnd(), index=seconds)
s_2 = pd.Series(rnd(), index=seconds)
s_3 = pd.Series(rnd(), index=seconds)
```

We use the same DatetimeIndex object we created in the previous recipe as the index for each of the Series. This lets pandas align each of the columns along the common index.

4. Create the DataFrame using a dictionary:

```
df = pd.DataFrame({"a": s_1, "b": s_2, "c": s_3})
```

The result is a DataFrame with a DatetimeIndex object of second resolution and three columns all with samples from a normal distribution:

	a	b	c
2016-01-01 00:00:00	-1.075846	2.547374	0.415277
2016-01-01 00:00:01	-0.457549	0.498616	-0.574556
2016-01-01 00:00:02	-0.644713	1.843763	0.388706
2016-01-01 00:00:03	-0.268644	1.186620	0.857448
2016-01-01 00:00:04	-0.590639	-0.421629	-1.244135
...	...	...	...
2016-01-01 00:01:35	-1.996332	-0.625641	-0.173174
2016-01-01 00:01:36	-1.502497	1.172084	-0.238044
2016-01-01 00:01:37	-0.572685	0.106536	0.499070
2016-01-01 00:01:38	-0.905668	0.382154	-0.221563
2016-01-01 00:01:39	0.548253	-1.149373	-0.257776

Figure 2.2: DataFrame with a DatetimeIndex object and three columns of random data

How it works...

pandas Series and DataFrames have several ways of being constructed. Series can take any array-like, iterable (for example, list or tuple), dictionary, or scalar value to be created. In this example, you created a NumPy array-like object called an array. We passed in the **`DatetimeIndex`** object created in the previous recipe, which is the same length as the array that was used to create the Series.

DataFrames accept any multidimensional structured or homogenous objects, be they iterable, dictionaries, or other DataFrames. In this example, you passed in a Python dictionary, which is a list of key-value pairs. Each

key represents the column name and each value represents the data for that column. DataFrames also accept an **`index`** argument in case the input data is not structured with an index. DataFrames can also accept a **`columns`** argument, which names the columns. Since the Series included indexes, pandas automatically used them to create the DataFrame index and aligned the values appropriately. If there were missing data for an index, pandas would still include the index but include **`nan`** as the value.

There's more...

A **`MultiIndex`** object is a multidimensional index that provides added flexibility to DataFrames. You can create a **`MultiIndex`** DataFrame from scratch or "reindex" an existing DataFrame to make it a **`MultiIndex`** DataFrame.

Building a MultiIndex DataFrame from scratch

Use the same **`MultiIndex`** object you created in the previous recipe:

```
tuples = [
    (pd.Timestamp("2023-07-10"), "WMT"),
    (pd.Timestamp("2023-07-10"), "JPM"),
    (pd.Timestamp("2023-07-10"), "TGT"),
    (pd.Timestamp("2023-07-11"), "WMT"),
    (pd.Timestamp("2023-07-11"), "JPM"),
    (pd.Timestamp("2023-07-11"), "TGT"),
]
midx = pd.MultiIndex.from_tuples(
    tuples,
    names=("date", "symbol")
)
```

Now that we have the index, we can create the DataFrame:

```
df_2 = pd.DataFrame(
    {
        "close":[158.11,144.64,132.55,158.20,146.61,134.86],
        "factor_1": [0.31, 0.24, 0.67, 0.29, 0.23, 0.71],
    },
```

```
    index=midx,
)
```

We create the DataFrame using a dictionary. The keys are the column names and the values are the data for the columns. Note that we use the **`index`** argument while passing in the **`MultiIndex`** object. The result is a DataFrame that contains records for each of the three symbols for each date:

		close	factor_1
date	**symbol**		
2023-07-10	**WMT**	158.11	0.31
	JPM	144.64	0.24
	TGT	132.55	0.67
2023-07-11	**WMT**	158.20	0.29
	JPM	146.61	0.23
	TGT	134.86	0.71

Figure 2.3: MultiIndex DataFrame

Reindexing an existing DataFrame with a MultiIndex object

It's common to add a **`MultiIndex`** object to a DataFrame. Let's consider an example of reindexing options data for a **`MultiIndex`** object:

1. Import the OpenBB Platform:

```
from openbb import obb
obb.user.preferences.output_type = "dataframe"
```

2. Download option chains using OpenBB:

```
chains = obb.derivatives.options.chains(
    "SPY",
    provider="cboe"
)
```

   The result is a DataFrame with a **`RangeIndex`** object starting at 0:

	contract_symbol	expiration	strike	option_type	open_interest	...	theta	vega	rho	last_trade_timestamp	dte
0	SPY240703C00445000	2024-07-03	445.0	call	2	...	0.0000	0.000	0.0000	2024-07-01 14:02:04	0
1	SPY240703P00445000	2024-07-03	445.0	put	5	...	0.0000	0.000	0.0000	2024-07-02 09:50:35	0
2	SPY240703C00450000	2024-07-03	450.0	call	5	...	0.0000	0.000	0.0000	2024-07-03 12:43:35	0
3	SPY240703P00450000	2024-07-03	450.0	put	30	...	0.0000	0.000	0.0000	2024-07-02 11:59:56	0
4	SPY240703C00455000	2024-07-03	455.0	call	0	...	0.0000	0.000	0.0000	NaT	0
...	...	...	...	...	...	...	...	...	...	...	...
8641	SPY261218P00810000	2026-12-18	810.0	put	0	...	-0.0692	0.000	-0.0001	NaT	898
8642	SPY261218C00815000	2026-12-18	815.0	call	0	...	-0.0077	1.309	0.9418	NaT	898
8643	SPY261218P00815000	2026-12-18	815.0	put	0	...	-0.0692	0.000	-0.0001	NaT	898
8644	SPY261218C00820000	2026-12-18	820.0	call	8	...	-0.0074	1.262	0.8956	2024-07-02 13:57:35	898
8645	SPY261218P00820000	2026-12-18	820.0	put	0	...	-0.0692	0.000	-0.0001	NaT	898

Figure 2.4: DataFrame with SPY options data

3. Options are derivatives that are frequently grouped by expiration date, strike price, option type, and any combination of those three. Using the **`set_index`** method takes the arguments in the list and uses those columns as indexes, converting **`RangeIndex`** into a **`MultiIndex`** object. In this example, we use the expiration date, strike price, and option type as the three indexes:

```
df_3 = chains.set_index(["expiration", "strike",
    "option_type"])
```

The result is a hierarchical DataFrame with a three-dimensional **`MultiIndex`**:

			contract_symbol	open_interest	...	last_trade_timestamp	dte
expiration	strike	option_type					
2024-05-28	444.0	call	SPY240528C00444000	1	...	2024-05-24 12:00:01	0
		put	SPY240528P00444000	498	...	2024-05-24 15:59:58	0
	445.0	call	SPY240528C00445000	3	...	2024-05-28 14:21:58	0
		put	SPY240528P00445000	201	...	2024-05-23 15:44:44	0
	446.0	call	SPY240528C00446000	0	...	NaT	0
...	...	...	...	...	...	...	...
2026-12-18	785.0	put	SPY261218P00785000	0	...	2024-04-03 09:36:19	934
	790.0	call	SPY261218C00790000	0	...	2024-05-28 13:07:31	934
		put	SPY261218P00790000	0	...	NaT	934
	795.0	call	SPY261218C00795000	4	...	2024-05-23 11:42:10	934
		put	SPY261218P00795000	0	...	NaT	934

Figure 2.5: A DataFrame with a three-dimensional MultiIndex

You can inspect the index using dot notation:

```
df_3.index
```

As you can see, the **`MultiIndex`** object is a list of tuples. Each tuple contains the elements of the **`MultiIndex`** object. We also have a property called **`names`**, which includes a list of the **`MultiIndex`** level names:

```
MultiIndex([('2023-07-12', 372.0, 'call'),
            ('2023-07-12', 372.0,  'put'),
            ('2023-07-12', 373.0, 'call'),
            ('2023-07-12', 373.0,  'put'),
            ('2023-07-12', 374.0, 'call'),
            ('2023-07-12', 374.0,  'put'),
            ('2023-07-12', 375.0, 'call'),
            ('2023-07-12', 375.0,  'put'),
            ('2023-07-12', 376.0, 'call'),
            ('2023-07-12', 376.0,  'put'),
            ...
            ('2025-12-19', 645.0, 'call'),
            ('2025-12-19', 645.0,  'put'),
            ('2025-12-19', 650.0, 'call'),
            ('2025-12-19', 650.0,  'put'),
            ('2025-12-19', 655.0, 'call'),
            ('2025-12-19', 655.0,  'put'),
            ('2025-12-19', 660.0, 'call'),
            ('2025-12-19', 660.0,  'put'),
            ('2025-12-19', 665.0, 'call'),
            ('2025-12-19', 665.0,  'put')],
           names=['expiration', 'strike', 'optionType'], length=7306)
```

Figure 2.6: Inspecting the MultiIndex object

See also

To learn more about the Series and DataFrame objects, take a look at the following resources:

- Documentation on pandas Series: **https://pandas.pydata.org/docs/reference/api/pandas.Series.html**
- Documentation on pandas DataFrames: **https://pandas.pydata.org/docs/reference/api/pandas.DataFrame.html**

Manipulating and transforming DataFrames

Before moving on to more advanced recipes, it's important to understand the fundamentals of working with data. DataFrames are the most common pandas data structures you'll work with. Despite the existence of hundreds of methods for DataFrame manipulation, only a subset of these are regularly used.

In this recipe, we will show you how to manipulate a DataFrame using the following common methods:

- Creating new columns using aggregates, Booleans, and strings
- Concatenating two DataFrames together
- Pivoting a DataFrame such as Excel
- Grouping data on a key or index and applying an aggregate
- Joining options data together to create straddle prices

Getting ready...

We'll start by importing the necessary libraries and downloading market price data:

1. Start by importing NumPy, pandas, and the OpenBB Platform:

```
import numpy as np
import pandas as pd
from openbb import obb
obb.user.preferences.output_type = "dataframe"
```

2. Load some stock price data to work with:

```
asset = obb.equity.price.historical(
    "AAPL",
    provider="yfinance"
)
benchmark = obb.equity.price.historical(
    "SPY",
    provider="yfinance"
)
```

How to do it...

Creating new columns is a common operation when dealing with DataFrames. Let's learn how to do this.

Creating new columns using aggregates, Booleans, and strings

Here's how to create new columns using aggregates, Booleans, and strings:

1. Start by renaming the columns so that they follow Python conventions:

```
columns = [
    "open",
    "high",
    "low",
    "close",
    "volume",
    "dividends",
    "splits",
]
asset.columns = columns
benchmark.columns = columns + ["capital_gain"]
```

2. Add a new column with values from an aggregate:

```
asset["price_diff"] = asset.close.diff()
benchmark["price_diff"] = benchmark.close.diff()
```

> *TIP*
>
> *When adding a new column to a DataFrame, the data you provide is automatically aligned by the DataFrame's index. This means that if you add a Series as a new column, pandas matches each value to its corresponding row in the DataFrame using the Series's index. If any index values in the DataFrame don't exist in the new Series, pandas will fill them with* **nan** *values, signifying missing data.*

3. Add a new column with a Boolean:

```
asset["gain"] = asset.price_diff > 0
```

```
benchmark["gain"] = benchmark.price_diff > 0
```

4. Add a new column with a string value:

```
asset["symbol"] = "AAPL"
benchmark["symbol"] = "SPY"
```

Three new columns of data are added to each DataFrame – **`returns`**, **`gain`**, and **`symbol`**:

date	open	high	low	close	adj_close	volume	dividends	splits	price_diff	gain	symbol
2020-08-17	114.001870	114.026427	111.939155	112.572701	112.572701	119561600	0.0	0.0	NaN	False	AAPL
2020-08-18	112.322219	113.940467	111.983343	113.510735	113.510735	105633600	0.0	0.0	0.938034	True	AAPL
2020-08-19	113.923294	115.082344	113.557410	113.653175	113.653175	145538000	0.0	0.0	0.142441	True	AAPL
2020-08-20	113.694908	116.290493	113.677717	116.175079	116.175079	126907200	0.0	0.0	2.521904	True	AAPL
2020-08-21	117.145048	122.650538	117.132773	122.161873	122.161873	338054800	0.0	0.0	5.986794	True	AAPL
...	...	...	...	...	...	...	...	...	...	...	...
2023-08-16	177.130005	178.539993	176.500000	176.570007	176.570007	46964900	0.0	0.0	-0.879990	False	AAPL
2023-08-17	177.139999	177.509995	173.479996	174.000000	174.000000	66062900	0.0	0.0	-2.570007	False	AAPL
2023-08-18	172.300003	175.100006	171.960007	174.490005	174.490005	61114200	0.0	0.0	0.490005	True	AAPL
2023-08-21	175.070007	176.130005	173.740005	175.839996	175.839996	46311900	0.0	0.0	1.349991	True	AAPL
2023-08-22	177.059998	177.677795	176.250000	177.229996	177.229996	41363946	0.0	0.0	1.389999	True	AAPL

Figure 2.7: Result of adding new columns to the asset DataFrame

5. Set a single value based on the aggregate of values:

```
asset_2 = asset.copy()
asset_2.at[
    asset_2.index[10],
    "volume"
] = asset_2.volume.mean()
```

You can inspect the value by using the **`iat`** method on the **`asset_2`** DataFrame:

```
asset_2.iat[10, 5]
```

The result is a scalar value representing the mean volume between indexes **`5`** and **`10`**.

Concatenating two DataFrames together

In the simplest case, concatenating two DataFrames together either stacks one on top of each other (row-wise concatenation) or lines them up next to each other (column-wise concatenation). You can control this through the **`axis`** argument.

Call the pandas **`concat`** method, leaving **`axis`** as the default (0 for row-wise concatenation):

```
pd.concat([asset, asset_2]).drop_duplicates()
```

Here are the concatenated DataFrames:

date	open	high	low	close	adj_close	volume	dividends	splits	price_diff	gain	symbol
2020-08-17	114.001870	114.026427	111.939155	112.572701	112.572701	119561600	0.0	0.0	NaN	False	AAPL
2020-08-18	112.322219	113.940467	111.983343	113.510735	113.510735	105633600	0.0	0.0	0.938034	True	AAPL
2020-08-19	113.923294	115.082344	113.557410	113.653175	113.653175	145538000	0.0	0.0	0.142441	True	AAPL
2020-08-20	113.694908	116.290493	113.677717	116.175079	116.175079	126907200	0.0	0.0	2.521904	True	AAPL
2020-08-21	117.145048	122.650538	117.132773	122.161873	122.161873	338054800	0.0	0.0	5.986794	True	AAPL
...	...	...	...	...	...	...	...	...	...	...	...
2023-08-17	177.139999	177.509995	173.479996	174.000000	174.000000	66062900	0.0	0.0	-2.570007	False	AAPL
2023-08-18	172.300003	175.100006	171.960007	174.490005	174.490005	61114200	0.0	0.0	0.490005	True	AAPL
2023-08-21	175.070007	176.130005	173.740005	175.839996	175.839996	46311900	0.0	0.0	1.349991	True	AAPL
2023-08-22	177.059998	177.677795	176.250000	177.229996	177.229996	41363946	0.0	0.0	1.389999	True	AAPL
2020-08-31	125.314885	128.674163	123.762935	126.748955	126.748955	212727600	0.0	4.0	4.157341	True	AAPL

Figure 2.8: Two DataFrames concatenated row-wise (stacked on top of each other)

The method takes a list of DataFrames to concatenate. In this example, we concatenated two identical DataFrames before dropping the duplicate values using the **`drop_duplicates`** method.

Pivoting a DataFrame such as Excel

Pivot tables are ubiquitous in Excel. They're used to aggregate data along a defined set of columns. They work the same way in pandas. In this example, we'll aggregate the returns using three methods – **`sum`**, **`mean`**, and **`std`**:

Call the pivot_table method:

```
pd.pivot_table(
    data=asset,
    values="price_diff",
    columns="gain",
    aggfunc=["sum", "mean", "std"]
)
```

The result is a pivoted DataFrame with `MultiIndex` column labels:

	sum		mean		std	
gain	False	True	False	True	False	True
returns	-745.20977	809.867065	-2.025027	2.076582	1.832028	1.773361

Figure 2.9: A pivoted DataFrame

Grouping data on a key or index and applying an aggregate

Grouping lets you aggregate different sections of your data. This is useful when you're working with market data and a single DataFrame contains prices for multiple assets:

1. Concatenate asset data and benchmark data into the same DataFrame:

   ```
   concated = pd.concat([asset, benchmark])
   ```

2. Group the resulting DataFrame by the `symbol` column, return the `adj_close` column, and apply the `ohlc` aggregate:

   ```
   concated.groupby("symbol").close.ohlc()
   ```

   The result is a DataFrame with `open`, `high`, `low`, and `close` properties for the asset symbol (**AAPL**) and the benchmark symbol (**SPY**):

	open	high	low	close
symbol				
AAPL	112.572701	196.185074	104.943108	177.229996
SPY	323.008636	466.563324	309.646606	438.149994

Figure 2.10: Grouped DataFrame

> *TIP*
>
> *The* `ohlc` *aggregate is a pandas* `Resampler`. `open` *is the first value of the group,* `high` *is the maximum value of the group,* `low` *is the minimum value of the group, and* `close` *is the last value of the group.*

Joining options data together to create straddle prices

DataFrame joins are similar to SQL joins. It combines two DataFrames based on a matching key. A great use case for joins is combining two DataFrames containing options chains to price straddles. A straddle is a complex options position made up of a long call and long put with the same strike and expiration:

1. Download option chains data using the OpenBB Platform:

```
chains = obb.derivatives.options.chains(
    "AAPL", provider="cboe")
```

2. Filter out the call options and the put options for a specific expiration:

```
expirations = chains.expiration.unique()
calls = chains[
    (chains.optionType == "call")
    & (chains.expiration == expirations[5])
]
puts = chains[
    (chains.optionType == "put")
    & (chains.expiration == expirations[5])
]
```

> *TIP*
>
> *The pandas* **`unique`** *method returns an array of unique values from the Series. In this case, it returns a unique set of expiration dates. From there, we select the expiration at index 5, which is sufficiently into the future and is where there will be trading activity.*

3. Set the index to the strike prices:

```
calls_strike = calls.set_index("strike")
puts_strike = puts.set_index("strike")
```

4. Then, join the call DataFrame on the put DataFrame using a left join:

```
joined = calls_strike.join(
    puts_strike,
    how="left",
    lsuffix="_call",
    rsuffix="_put"
)
```

Since both DataFrames have a column named **`lastPrice`**, we add the **`lsuffix`** and **`rsuffix`** arguments to distinguish the **`lastPrice`** columns from each other.

5. Use only the price columns from the joined DataFrame:

```
prices = joined[["last_trade_price_call",
     "last_trade_price_put"]]
```

6. Sum up the call and put prices by using the **`axis`** argument in the **`sum`** method:

```
prices["straddle_price"] = prices.sum(axis=1)
```

The result is a DataFrame with the strike prices as the index, and the call, put, and straddle prices as columns for each strike:

strike	last_trade_price_call	last_trade_price_put	straddle_price
100.0	0.0	0.0	0.0
105.0	0.0	0.0	0.0
110.0	0.0	0.0	0.0
115.0	0.0	0.0	0.0
120.0	0.0	0.0	0.0
...	...	...	...
245.0	0.0	0.0	0.0
250.0	0.0	0.0	0.0
255.0	0.0	0.0	0.0
260.0	0.0	0.0	0.0
265.0	0.0	0.0	0.0

Figure 2.11: DataFrame containing call, put, and straddle prices

How it works...

In this recipe, we covered four ways of manipulating DataFrames.

DataFrames supports column addition via the **`df["new_col"] = value`** assignment syntax. For aggregates, methods such as **`groupby`** and **`agg`** compute summary metrics, which can be joined to the original DataFrame. By

using Boolean indexing, such as **`df["new_col"] = df["existing_col"] > threshold`**, we can create new columns based on conditions.

The **`pivot_table`** method reshapes data, providing a multidimensional analysis tool similar to pivot tables in Excel. By specifying the **`index`** and **`columns`** parameters, we can set row and column keys, respectively. The **`values`** parameter denotes which DataFrame columns to aggregate, and **`aggfunc`** defines the aggregation function, such as **`sum`** or **`mean`**. This method then constructs a new DataFrame, where rows and columns are determined by unique combinations of key values.

The **`groupby`** method enables data segmentation based on column values. By passing one or multiple column names to **`groupby`**, we create a **`GroupBy`** object, segmenting the data. This object allows aggregation, transformation, or filtering of these groups using methods such as **`sum`**, **`mean`**, or **`apply`**. The result is a DataFrame with an index based on the grouping columns and aggregated or transformed values.

Finally, we covered the join method, which combines two DataFrames based on their indexes. Specifying the **`on`** parameter can alter the default behavior, using a particular column as the join key. The **`how`** parameter determines the type of join: **`left`**, **`right`**, **`inner`**, or **`outer`**. This method appends columns from the joining DataFrame to the calling DataFrame based on matching index or column values.

There's more...

The pandas **`groupby`** method is one of the most used transforms since it lets you aggregate and perform analysis on different chunks of data in the same DataFrame. Here are a few more ways to use it.

Grouping by multiple columns

`groupby` can be used with multiple columns, providing a detailed view of your data. This is useful when you want to group data based on a combination of values from different columns. For example, let's say you want to aggregate the open interest for each contract:

```
(
    chains
    .groupby(
        ["option_type", "strike", "expiration"]
    )
    .open_interest
    .sum()
)
```

The result is a Series with a **`MultiIndex`** object composed of **`optionType`**, **`strike`**, and **`expiration`**. The values are the sum of the **`openInterest`** aggregate for each group:

```
option_type  strike  expiration
call         5.0     2024-03-15      2.0
                     2024-06-21    341.0
                     2024-09-20    128.0
             10.0    2024-03-15      0.0
                     2024-06-21     12.0
                                    ...
put          330.0   2024-08-16      0.0
             340.0   2024-08-16      0.0
             350.0   2024-08-16      0.0
             360.0   2024-08-16      0.0
             370.0   2024-08-16      0.0
Name: open_interest, Length: 2086, dtype: float64
```

Figure 2.12: Option chains grouped by option type, strike, and expiration with the openInterest aggregate

Applying different methods to different columns

You can use **`agg`** to apply specific methods to specific columns after grouping. For example, let's say you want to find the maximum **`lastPrice`** and the total **`openInterest`** for each **`optionType`**:

```
(
    chains
    .groupby(
        ["option_type", "strike", "expiration"]
    )
    .agg({
        "last_trade_price": "max",
        "open_interest": "sum"
```

```
    })
)
```

This provides a DataFrame with **`option_type`**, **`strike`**, and **`expiration`** as the indexes and two columns: one with the maximum **`last_price`** and the other with the sum of **`open_interest`** for each option type. It offers a snapshot of the highest option price and total open interest for the call and put options:

			last_trade_price	open_interest
option_type	strike	expiration		
call	5.0	2024-06-21	185.05	31
		2024-07-19	164.00	0
		2024-08-16	182.23	1
		2024-09-20	179.30	2
		2024-10-18	172.06	0
...	...	...	...	...
put	370.0	2024-08-16	0.00	0
		2024-10-18	0.00	0
		2024-11-15	187.00	0
		2025-03-21	0.00	0
	380.0	2025-03-21	189.89	0

Figure 2.13: Maximum price and aggregate open interest for each combination of option type, strike, and expiration

Applying custom functions

You can apply your own functions using the **`apply`** method. For example, let's say you want to calculate the average spread (difference between **`ask`** and **`bid`**) for each group of **`optionType`**:

```
(
    chains
    .groupby(["option_type"])
    .apply(lambda x: (x["ask"] - x["bid"]).mean(),        include_groups=False
)
```

The result is a Series with **`option_type`** as the index, where the values represent the average spread for each option type:

```
option_type
call    1.274094
put     1.260757
dtype: float64
```

Figure 2.14: Average spread for calls and puts

> *TIP*
>
> *A lambda function can't be defined using the* **`def`** *keyword. It can accept any number of input arguments and returns any number of outputs. Lambda functions must exist on a single line and are useful for logic that can be written concisely.*

Grouping and transforming data

The **`transform`** method can be used after **`groupby`** to return a Series with the same shape as the original, where each group is replaced with the value from a method applied to that group. For example, let's say you want to compute the z-score of **`lastPrice`** within each **`expiration`** date:

```
(
    chains
    .groupby("expiration")
    .last_trade_price
    .transform(lambda x: (x - x.mean()) / x.std())
)
```

This results in a Series of the same length as **`chains`**, where **`lastPrice`** is replaced with its z-score value within each **`expiration`** date. The z-score can be useful for comparisons as it puts all the prices on the same scale relative to their specific expiration date group:

```
0       3.552798
1      -0.646571
2       3.226523
3      -0.646871
4       3.142709
          ...
1855    1.456281
1856   -0.917640
1857    1.613997
1858   -0.942438
1859   -1.048326
Name: lastPrice, Length: 1860, dtype: float64
```

Figure 2.15: Creating a standardized price within each expiration

See also

For more on option straddles, Investopedia offers a high-quality description. To learn more about the methods that were used in this recipe, take a look at the following documentation:

- Understanding option staddles: **https://www.investopedia.com/terms/s/straddle.asp**
- Documentation on concatenating pandas data structures: **https://pandas.pydata.org/docs/reference/api/pandas.concat.html**
- Documentation on pivoting pandas DataFrames: **https://pandas.pydata.org/docs/reference/api/pandas.DataFrame.pivot.html**
- Documentation on grouping pandas DataFrames: **https://pandas.pydata.org/docs/reference/api/pandas.DataFrame.groupby.html**
- Documentation on joining pandas DataFrames: **https://pandas.pydata.org/docs/reference/api/pandas.DataFrame.join.html**

Examining and selecting data from DataFrames

Once you've loaded, manipulated, and transformed data in DataFrames, the next step is retrieving the data from DataFrames. This is where indexing and selecting data come into play. This functionality allows you to ac-

cess data using methods such as **`iloc`** and **`loc`** and techniques such as Boolean indexing or query functions. These methods can target data based on its position, labels, or condition based on whether you're after a specific row, column, or combination. Inspection enables potential issues to be identified, such as missing values, outliers, or inconsistencies, that can affect analysis and modeling. Additionally, an initial inspection provides insights into the nature of data, helping determine appropriate preprocessing steps and analysis methods.

How to do it...

Let's start by downloading stock price data:

1. Start by importing pandas and the OpenBB Platform:
```
import pandas as pd
from openbb import obb
obb.user.preferences.output_type = "dataframe"
```
2. Then, load some data:
```
df = obb.equity.price.historical(
    "AAPL",
    start_date="2021-01-01",
    provider="yfinance"
)
```
3. Display the first 5 records:
```
df.head(5)
```
4. Display the last 5 records:
```
df.tail(5)
```
5. Return the DataFrame as a NumPy array:
```
df.values
```
6. Get descriptive statistics:
```
df.describe()
```
7. Rename the columns so that they follow Python conventions:

```
df.columns = [
    "open",
    "high",
    "low",
    "close",
    "volume",
    "dividends",
    "splits",
]
```

8. Select one column of data:

```
df["close"]
```

IMPORTANT NOTE

When selecting one column from a DataFrame, the return type is a Series. If you try to apply a DataFrame method on a Series, this results in an exception. To select one column while returning a DataFrame instead of a Series, slice the DataFrame using a list by using **`df[["close"]]`.**

9. Select rows by index:

```
df[0:3]
```

IMPORTANT NOTE

When slicing Python objects using indexes, the result is not inclusive of the last index. For example, when slicing using indexes 0:3, the values at index location 0, 1, and 2 are returned, but not the value at index location 3.

10. pandas is smart enough to turn strings into **`DatetimeIndex`** values and use them to return a range of data between two dates:

```
df.index = pd.to_datetime(df.index)
df["2021-01-02":"2021-01-11"]
```

IMPORTANT NOTE

> *When slicing Python objects using labels, the result is inclusive of the last label. For example, when slicing using labels "2021-01-02":"2021-01-11", the value at label location "2021-01-11" is included in the return value.*

Selection by label using loc

The **`loc`** method is a method that's used for data selection based on label-based indexing. It allows data to be selected by row, column, or both, using labels of rows or list-like objects. This method is useful when the index of a DataFrame is a label other than a numerical series:

1. Selecting a single row transposes the Series and puts the DataFrame columns as row labels:
```
df.loc[df.index[0]]
```
2. Selecting a single row and single column returns a scalar:
```
df.loc[df.index[0], "close"]
```
3. Selecting a range of rows and a list of columns returns a subsection of the data:
```
df.loc[df.index[0:6], ["high", "low"]]
```
4. Selecting a range of labels and a list of columns returns a subsection of the data:
```
df.loc["2021-01-02":"2021-01-11", ["high", "low"]]
```

Selection by position using iloc

The pandas **`iloc`** method is a data selection method that's used for indexing based on integer locations. **`iloc`** works independently of the DataFrame's index labels:

1. Selecting a single row by index location transposes the Series and puts the DataFrame columns as row labels:
```
df.iloc[3]
```
2. Selecting a range of rows by index and a range of columns by index returns a subset of the data:

```
df.iloc[3:5, 0:2]
```

3. Selecting a combination of specific rows and columns by index works too:

```
df.iloc[[1, 2, 4], [0, 2]]
```

Selection by Boolean indexing

Boolean indexing involves generating a Boolean Series that corresponds to the rows in the DataFrame, where **`True`** indicates that the row meets the condition and **`False`** denotes it does not. By passing this Boolean Series to the DataFrame, it returns only the rows where the condition is **`True`**, thereby allowing for conditional selection and manipulation of data:

1. Inspect where a condition is True for each row:

```
df.close > df.close.mean()
```

2. Using single-column values to select data:

```
df[df.close > df.close.mean()]
```

3. Use the result to return all the rows and the data at column index location 0:

```
df[df.close > df.close.mean()].iloc[:, 0]
```

4. Use label-based indexing with Boolean indexing to create query-like slicing:

```
df.loc[
    (df.close > df.close.mean())
    & (df.close.mean() > 100)
    & (df.volume > df.volume.mean())
]
```

How it works…

The **`loc`** method is for label-based data selection within a DataFrame or Series. We can reference rows and columns using their labels, defining

both the row index and column name in **`df.loc[row, column]`** format. It supports slicing, as we saw previously, enabling multiple rows and columns to be selected based on label ranges.

The **`iloc`** method is for integer-location-based indexing. We use the index positions referencing rows and columns in **`df.iloc[row, column]`** format. It also supports slicing, allowing multiple rows and columns to be selected based on integer ranges. Unlike **`loc`**, **`iloc`** disregards index or column labels and strictly operates on integer positions.

Boolean indexing allows for data filtering based on condition evaluations. Within a DataFrame or Series, a condition that's applied to a column or the entire structure returns a Boolean array, where **`True`** values indicate rows meeting the condition. By placing this Boolean array within the DataFrame's square brackets – for example, **`df[boolean_array]`** – only the rows with **`True`** values are extracted. Multiple conditions can be combined using the **`&`** (and), **`|`** (or), and **`~`** (not) operators. Additionally, the **`query`** method can be used as an alternative to achieve the same result in a more readable syntax. Boolean indexing can return results from full DataFrames considerably faster than looping or enumerating through DataFrame rows.

There's more...

Indexing is one of the most important aspects of working with pandas. Let's look at some advanced examples of how to slice data.

Partial string indexing

You can select data using a **`DatetimeIndex`** object based on a partial string:

```
df["2023"]
```

The preceding code will return a DataFrame containing all records from 2023. The resulting DataFrame maintains the same columns as the original DataFrame but only includes rows where the index falls within 2023.

You can also include the month:

```
df["2023-07"]
```

The preceding code will return a DataFrame with all records from July 2023.

Using at for fast access to a scalar

If we want to quickly access the value of a specific cell, we can use the **`at`** accessor. For example, to get the **`close`** price on July 12, 2023, we can run the following command:

```
df.at["2023-07-12", "close"]
```

The result is the closing price on July 12, 2023. The **`at`** accessor provides a fast way to access a single value at a specific row-column pair.

Using nsmallest and nlargest

Pandas provides handy methods to get rows with the smallest or largest values in a column. For instance, to get the five rows with the highest volume, we can use the following command:

```
df.nlargest(5, "volume")
```

The result is a DataFrame with the five rows with the highest **`volume`** values.

Using the query method to query DataFrames

The **`query`** method allows you to query a DataFrame in a more readable way. For instance, to get all rows where the **`close`** price is higher than the **`open`** price, you can use the following command:

```
df.query("close > open")
```

The resulting DataFrame will contain all rows where the **`close`** price was higher than the **`open`** price.

See also

Indexing can be a complex topic that takes practice to master. This recipe covers many use cases that are common in algorithmic trading. For more details, take a look at the following documentation on indexing and slicing pandas DataFrames: https://pandas.pydata.org/docs/user_guide/indexing.html.

Calculating asset returns using pandas

Returns are integral to understanding the performance of a portfolio. There are two types: simple returns and compound (or log) returns.

Simple returns, which are calculated as the difference in price from one period to the next divided by the price at the beginning of the period, are beneficial in certain circumstances. They aggregate across assets, meaning the simple return of a portfolio is the aggregate of the returns of the individual assets, weighted according to their proportions. This trait makes simple returns practical for comparing assets and evaluating portfolio performance over short-term intervals.

Simple returns are defined as follows:

$$R_t = \frac{P_t - P_{t-1}}{P_{t-1}} = \frac{P_t}{P_{t-1}} - 1$$

On the other hand, **compound returns**, which are calculated using the natural logarithm of the price-relative change, are additive over time. This quality makes them suitable for multi-period analyses as the compound return for a given period is the sum of the log returns within that period.

Compound returns are defined as follows:

$$r_t = \log\left(\frac{P_t}{P_{t-1}}\right) = \log(P_t) - \log\left(P_{t-1}\right)$$

Compound returns are generally preferred in practice since they have mathematical properties that make them easier to use in analytics. They're less influenced by extreme values, which might skew the analysis. A major price change in a single period will have a large effect on simple returns but a moderated effect on log returns.

The additive nature of compound returns across time is mathematically convenient. If you need to calculate the total return over a sequence of periods, you simply sum up the log returns, whereas for simple returns, you would need to calculate the product of (1 plus each period's return) and then subtract one.

Given a daily compounded return, r_t, it is straightforward to solve back for the corresponding simple net return, R_t:

$$R_t = e^{r_t} - 1$$

The difference between simple and log returns becomes evident when we look into the statistical properties. If we assume that asset prices follow a log-normal distribution, which may not always be the case, the log returns would be normally distributed. This assumption is beneficial because normal distribution is a cornerstone in statistical modeling. For daily or intraday data, the difference between simple and log returns is usually minimal, with log returns generally being slightly smaller.

How to do it...

Let's start building returns using stock data from the OpenBB Platform:

1. Import pandas, NumPy, and the OpenBB Platform:

```
import numpy as np
import pandas as pd
from openbb import obb
obb.user.preferences.output_type = "dataframe"
```

2. Download stock price data for AAPL:

```
data = obb.equity.price.historical(
    "AAPL",
```

```
    provider="yfinance"
)
```

3. Select the close data:

```
df = data.loc[:, ["close"]]
```

4. Add a new column with simple returns and compound returns using the adjusted closing price:

```
df["simple"] = df["close"].pct_change()
df["compound"] = np.log(
    df["close"] / df["close"].shift()
)
```

The resulting DataFrame now has two additional columns:

date	close	simple	compound
2023-05-30	177.300003	NaN	NaN
2023-05-31	177.250000	-0.000282	-0.000282
2023-06-01	180.089996	0.016023	0.015896
2023-06-02	180.949997	0.004775	0.004764
2023-06-05	179.580002	-0.007571	-0.007600
...	...	...	...
2024-05-21	192.350006	0.006857	0.006834
2024-05-22	190.899994	-0.007538	-0.007567
2024-05-23	186.880005	-0.021058	-0.021283
2024-05-24	189.979996	0.016588	0.016452
2024-05-28	189.990005	0.000053	0.000053

Figure 2.16: The resulting DataFrame with daily simple and compound returns

How it works...

First, we imported NumPy for numerical operations, pandas for data manipulation, and the OpenBB Platform for financial data retrieval.

Then, we used the OpenBB Platform's **`stocks.load`** method to fetch the stock price data for AAPL. The data was stored in a DataFrame.

Next, we created a new DataFrame containing only the **`close`** column. We did this using the **`loc`** function, a label-based data selector.

After, we used the **`pct_change`** method to calculate simple returns, which represent the percentage change in the adjusted close price from the previous day. The result was stored in a new column called **`simple`**.

For compound returns, we calculated the ratio of the current day's adjusted close price to the previous day's price using the **`shift`** method. This shifted the DataFrame index by one period. Then, we took the natural logarithm of these ratios with **`log`**, reflecting the formula for compound returns. The results were stored in another new column called **`compound`**.

There's more...

The **`pct_change`** method accepts a **`periods`** argument that will shift the input data by that number before computing the simple return. This is useful for computing returns that are multiple periods, such as 3 days:

```
df["close"].pct_change(periods=3)
```

The result is a new Series, where each element represents a three-period simple return. Note that the first three elements of the resulting Series will be **`nan`** because there aren't enough prior periods for the calculation. (We'll learn how to fill in missing data in the *Addressing Missing Data Issues* recipe in this chapter.)

You can also do the same with compound returns by passing an argument to the **`shift`** method:

```
np.log(df["close"] / df["close"].shift(3))
```

The **`pct_change`** method also accepts a **`freq`** parameter, which lets you resample the return to an aligned period. For example, instead of passing in

22 to the **`periods`** argument (since not all months have 22 trading days), you can pass in **`ME`**:

```
df.index = pd.to_datetime(df.index)
df["close"].pct_change(freq="ME").dropna()
```

The result is a DataFrame with monthly returns:

```
date
2020-07-31    0.109736
2020-08-31    0.216569
2020-09-30   -0.102526
2020-11-30    0.096400
2020-12-31    0.114574
2021-03-31   -0.044135
2021-04-30    0.076218
2021-06-30    0.102028
2021-08-31    0.044925
2021-09-30   -0.068037
2021-11-30    0.111313
2021-12-31    0.074228
2022-01-31   -0.015712
2022-02-28   -0.054066
2022-03-31    0.057473
2022-05-31   -0.056352
2022-06-30   -0.081430
2022-08-31   -0.025210
2022-09-30   -0.120977
2022-10-31    0.109551
2022-11-30   -0.033027
2023-01-31    0.153674
2023-02-28    0.023183
2023-03-31    0.118649
2023-05-31    0.046613
2023-06-30    0.094330
Name: Adj Close, dtype: float64
```

Figure 2.17: Series with simple returns resampled to a monthly frequency

See also

For more on the **`pct_change`** method, see the following documentation: **https://pandas.pydata.org/docs/reference/api/pandas.DataFrame.pct_change.html**.

Measuring the volatility of a return series

Volatility plays an integral role in finance, serving as a key indicator of risk linked to a particular asset. A higher degree of volatility suggests a greater risk associated with the asset as it indicates more significant price changes and, therefore, a less predictable investment outcome.

Standard deviation is widely used as the measure of asset return volatility. It statistically quantifies the dispersion of asset returns from their mean, thus providing an effective metric for risk. When asset returns exhibit a larger standard deviation, it signifies more pronounced volatility, pointing to a higher risk level. Conversely, a lower standard deviation implies that the asset returns are more stable and less likely to deviate significantly from their average, indicating a lower risk.

The standard deviation's value as a risk measure extends beyond its ability to quantify risk alone. It is a common component in calculating risk-adjusted returns that provides a more nuanced evaluation of investment performance by considering the risk taken to achieve the returns. The Sharpe ratio, for instance, is a popular risk-adjusted return measure that divides the excess return of an investment (over the risk-free rate) by its standard deviation.

How to do it...

We'll use AAPL's historic stock price to compute the volatility. Here's how it's done:

1. Import the necessary libraries:

```
import numpy as np
import pandas as pd
from openbb import obb
obb.user.preferences.output_type = "dataframe"
```

2. Load stock price data from the OpenBB Platform:

```
df = obb.equity.price.historical(
    "AAPL",
    start_date="2020-01-01",
    provider="yfinance"
)
```

3. Grab the series of adjusted close prices:

```
close = df["close"]
```

4. Compute the daily simple return:

```
returns = close.pct_change()
```

5. Calculate the daily standard deviation of returns:

```
std_dev = returns.dropna().std()
```

6. Annualize the standard deviation assuming 252 trading days in a year:

```
annualized_std_dev = std_dev * np.sqrt(252)
```

The result is a float that represents the annualized volatility over the entire period.

How it works…

The multiplication that's done by the square root of 252 in the calculation of standard deviation is based on the concept of scaling in statistics. Scaling is used when transferring a data point from one timescale to another. The number 252 is used because there are typically 252 trading days in a year. In statistical terms, variance, which is the square of standard deviation, is additive over time. This means that if you want to calculate the variance over a certain period, you can add up the variances for each sub-period. However, the standard deviation is not additive, but its square (that is, variance) is. To convert daily standard deviation into annual standard deviation, we must square it to get the variance, multiply this variance by 252 (the typical number of trading days in a year) to annualize, and then take the square root to revert to standard deviation. This process ensures we respect the additive property of variance while transitioning from daily to annual measure. By taking the square root of 252, we're essentially projecting the daily volatility over a yearly timeframe under the assumption of independent daily returns.

There's more…

Depending on the frequency of returns, we can annualize the volatility measure to match. For example, if you're looking over a long period, you may be dealing with monthly or quarterly returns. Alternatively, you may be working with daily data and want to resample to a monthly or quarterly period:

```
close.index = pd.to_datetime(close.index)
(
    close
    .pct_change(freq="ME")
    .dropna()
    .std()
    * np.sqrt(12)
)
```

You can pass the **`freq`** argument to the **`pct_change`** method to generate monthly returns. Then, instead of multiplying the standard deviation by the square root of 252, you can multiply it by the square root of **`12`**.

Similarly, for quarters, you can pass the **`freq`** argument and multiply the standard deviation by the square root of 4:

```
(
    close
    .pct_change(freq="QE")
    .dropna()
    .std()
    * np.sqrt(4)
)
```

In both cases, the result is a float that represents the annualized volatility.

Looking at historical price changes, rather than just one moment, is important to get a broader picture of how much and how often the price has moved in the past. We'll use the pandas **`rolling`** method to plot the annualized volatility through time:

```
(
    close
    .pct_change()
    .rolling(window=22)
    .std()
    * np.sqrt(252)
).plot()
```

This results in the following plot:

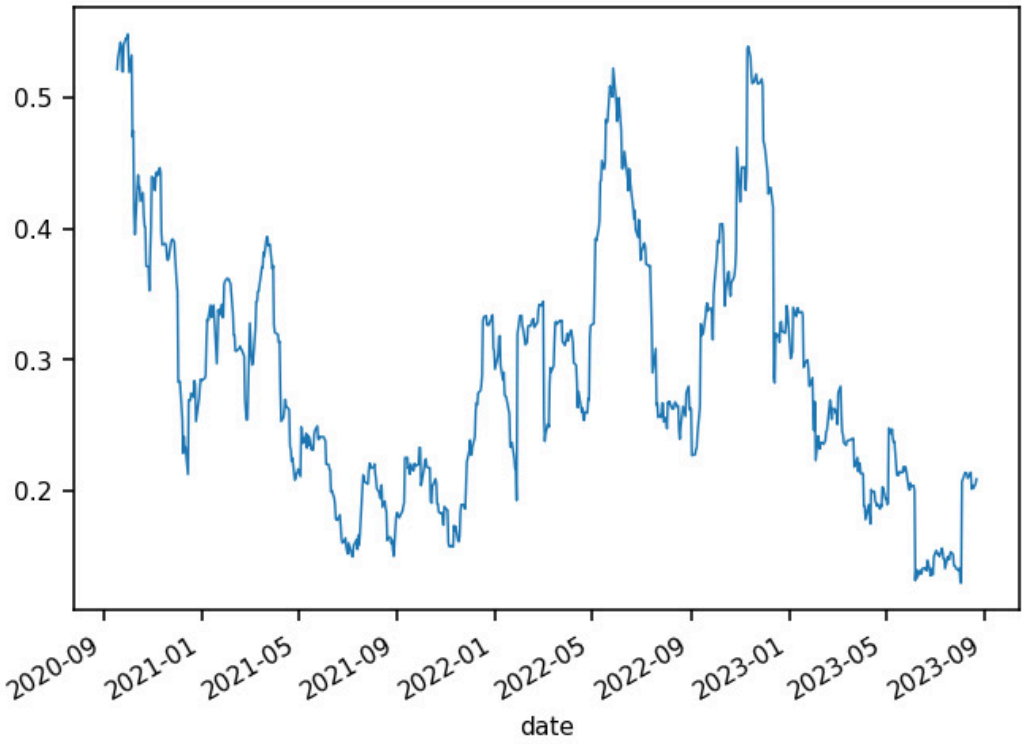

Figure 2.18: Rolling annualized volatility based on a 22-day lookback window

> *TIP*
>
> *Rolling a method in pandas involves taking a range of data, applying a function or method that returns a single value, moving that range of data forward one step, applying the function, and repeating this. The result is a method being applied over a window of data that extends through the entire range of data.*

See also

Volatility is a critical part of algorithmic trading. To learn more about volatility, take a look at the following article, which describes how investors think about volatility for risk management: https://www.investopedia.com/terms/v/volatility.asp.

Generating a cumulative return series

Cumulative returns quantify the total change in the value of an investment over a specific period. To compute the cumulative return of a series of simple single-period returns, you need to add 1 to each return, then multiply these results together, and finally subtract 1 from the product. Recall the formula for the simple return:

$$R_t = \frac{P_t - P_{t-1}}{P_{t-1}} = \frac{P_t}{P_{t-1}} - 1$$

Upon writing $\left(\frac{P_t}{P_{t-2}}\right) = \left(\frac{P_t}{P_{t-1}}\right)\left(\frac{P_{t-1}}{P_{t-2}}\right)$, the two-period return can be expressed as follows:

$$R(2) = \left(\frac{P_t}{P_{t-1}}\right)\left(\frac{P_{t-1}}{P_{t-2}}\right) - 1$$

$$= (1 + R_t)(1 + R_{t-1}) - 1$$

To compute the cumulative return of a series of continuously compounded single-period returns, you need to sum up all the single-period returns. This is because, in the case of continuously compounded returns, the logarithmic returns are additive. To illustrate this, consider a two-period compound return, which is defined as follows:

$$r_t(2) = \ln(1 + R_t(2)) = \ln\left(\frac{P_t}{P_{t-1}}\right) = \ln(P_t) - \ln(P_{t-1}) = p_t - p_{t-1}$$

Taking the exponent of both sides and rearranging the preceding formula gives us the following output:

$$p_t = P_{t-2}\, e^{r_t(2)}$$

Here, $r_t(2)$ is the compounded growth rate of prices between periods $t - 2$ and t. Using P t _ P t−2 = (P t _ P t−1)(P t−1 _ P t−2) and the fact that $\ln(xy) = \ln(x) + \ln(y)$, we get the following:

$$r(2) = \ln\left(\left(\frac{P_t}{P_{t-1}}\right)\left(\frac{P_{t-1}}{P_{t-2}}\right)\right)$$

$$= ln\left(\frac{P_t}{P_{t-1}}\right) + ln\left(\frac{P_{t-1}}{P_{t-2}}\right)$$

$$= r_t + r_{t-1}$$

Getting ready…

We assume that you've followed the instructions from the previous recipes and have a Series called **`close`** with a **`DatetimeIndex`** object as the index.

How to do it…

We'll use pandas to compute the cumulative sum of simple and compound returns.

The cumulative sum of simple returns

We'll start with the cumulative sum of simple returns:

1. Compute the single period daily simple returns:
   ```
   returns = close.pct_change()
   ```
2. Replace all **`nan`** values with zeros by using NumPy's **`isnan`** method as a mask:
   ```
   returns[np.isnan(returns)] = 0
   ```
   Note that this is equivalent to using pandas:
   ```
   returns.fillna(0.0, inplace=True)
   ```
3. Now, add **`1`** to each of the daily simple returns:
   ```
   returns += 1
   ```
4. Use the **`cumprod`** method to build the cumulative product of returns and subtract **`1`**:
   ```
   cumulative_returns = returns.cumprod() - 1
   ```
5. Plot the result:
   ```
   cumulative_returns.plot()
   ```
   The result is the following plot:

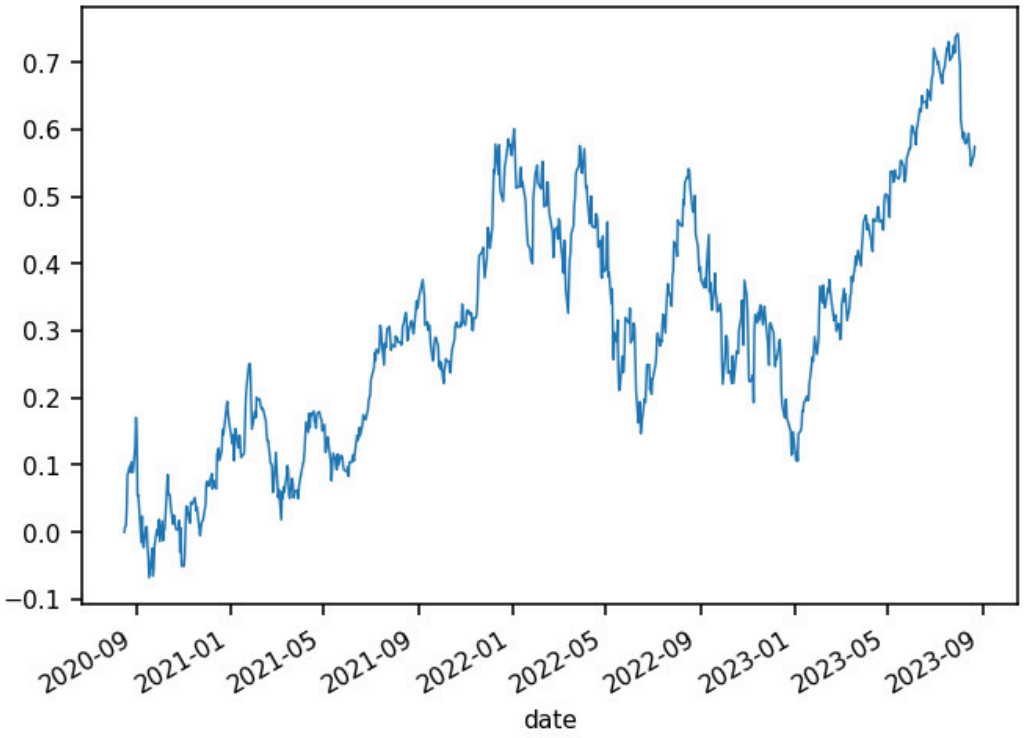

Figure 2.19: Cumulative simple returns of AAPL

The cumulative sum of compound returns

Next, we'll build the cumulative sum of compound returns:

1. Compute the single-period compound returns:

```
log_returns = np.log(close / close.shift())
```

2. Use the pandas **`cumsum`** method to build the cumulative product of returns:

```
cumulative_log_returns = log_returns.cumsum()
```

3. Plot the result:

```
cumulative_log_returns.plot()
```

The result is the following plot:

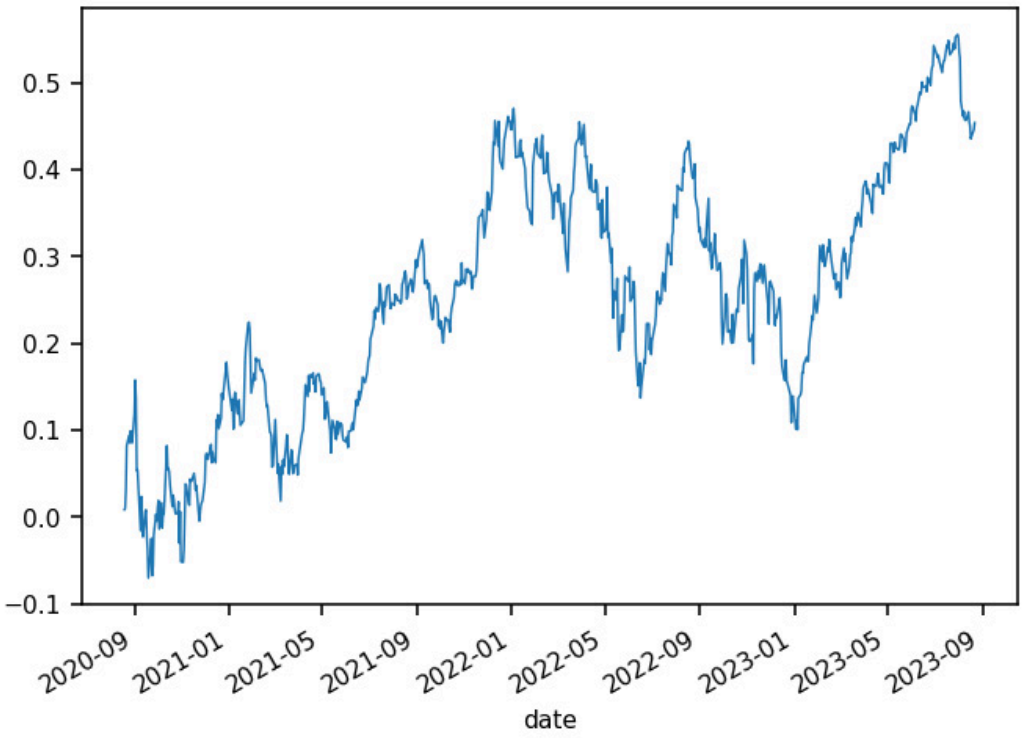

Figure 2.20: Cumulative compound returns of AAPL

Note that simple returns and compound returns will not be equal. Compound returns take into consideration returns earned on prior returns, while simple returns do not.

How it works...

`isnan` checks each value in the **`returns`** Series to see if it is a **`nan`** value. **`returns[np.isnan(returns)] = 0`** takes all the positions in the **`returns`** Series where the value is **`nan`** and replaces those **`nan`** values with **`0`**. To compute cumulative simple returns, we first need to add **`1`** to every single period return. This can be accomplished by incrementing every value in the returns Series by **`1`** with **`returns += 1`**. Finally, we generate the cumulative product using the **`cumprod`**. It calculates the cumulative product of the Series, meaning it multiplies the values in the Series together and keeps track of the result. For example, if you have a Series with the numbers 1, 2, 3, and 4, the **`cumprod`** method will create a new Series with the values 1, *2 (1 x 2)*, *6 (1 x 2 x 3)*, and *24 (1 x 2 x 3 x 4)*. Finally, we subtract **`1`** from each value in the cumulative series, compute the compound returns, and call **`cumsum`** to generate the cumulative sum of the values. Instead of multiplying the values together via **`cumprod`**, **`cumsum`** adds them.

See also

While computing returns may seem trivial, it's important to understand the difference between simple returns and compound returns. You can learn more about them, along with the methods we used to compute them, by taking a look at the following documentation:

- A deep dive into the mathematics behind simple and compound returns (and a lot more): **https://bookdown.org/compfinezbook/introcompfinr/AssetReturnCalculations.html#continuously-compounded-returns**
- Documentation on the pandas `cumsum` method: **https://pandas.pydata.org/docs/reference/api/pandas.Series.cumsum.html**
- Documentation on the pandas `cumprod` method: **https://pandas.pydata.org/docs/reference/api/pandas.Series.cumprod.html**

Resampling data for different time frames

Two types of resampling are upsampling, where data is converted into a higher frequency (such as daily data to hourly data), and downsampling, where data is converted into a lower frequency (such as daily data to monthly data). In financial data analysis, resampling can help in various ways. For instance, if you have daily stock prices, you can resample this data to calculate monthly or yearly average prices, which can be useful for long-term trend analysis. A common use case is when aligning trade and quote data. There are a lot more quotes than trades – often an order of magnitude more – and we may need to align the open, high, low, and closing quote prices to the open, high, low, and closing trade data. Since the quotes and trades will have different timestamps, resampling to a 1-second resolution is a great way to align these disparate data sources.

How to do it...

We'll work on resampling stock price data from one period to another:

1. Import the necessary libraries:

```
import numpy as np
import pandas as pd
from openbb import obb
obb.user.preferences.output_type = "dataframe"
```

2. Use the OpenBB Platform to download intraday data at 1-minute intervals:

```
df = obb.equity.price.historical(
    "AAPL",
    interval="1m",
    provider="yfinance"
)
```

3. Resample the 1-minute resolution adjusted close data to hourly:

```
resampled = df.resample(rule="h")["close"]
```

4. Grab the first value within the resampled time interval:

```
resampled.first()
```

Here's the output:

```
date
2023-07-12 09:00:00    189.910004
2023-07-12 10:00:00    190.619995
2023-07-12 11:00:00    189.716003
2023-07-12 12:00:00    188.897293
2023-07-12 13:00:00    189.429993
                          ...
2023-07-18 11:00:00    193.489899
2023-07-18 12:00:00    193.130005
2023-07-18 13:00:00    192.660004
2023-07-18 14:00:00    192.985001
2023-07-18 15:00:00    193.669907
Freq: H, Name: Adj Close, Length: 151, dtype: float64
```

Figure 2.21: Resampled Series with the first value in the interval

5. Now, grab the last value:

```
resampled.last()
```

Here's the output:

```
date
2023-07-12 09:00:00    190.710007
2023-07-12 10:00:00    189.669907
2023-07-12 11:00:00    188.639893
2023-07-12 12:00:00    189.360992
2023-07-12 13:00:00    189.919998
                           ...
2023-07-18 11:00:00    193.199997
2023-07-18 12:00:00    192.520004
2023-07-18 13:00:00    192.985001
2023-07-18 14:00:00    193.789902
2023-07-18 15:00:00    193.729996
Freq: H, Name: Adj Close, Length: 151, dtype: float64
```

Figure 2.22: Resampled Series with the last value in the interval

6. Compute the mean value within the resampled time interval:

   ```
   resampled.mean()
   ```

   Here's the output:

   ```
   date
   2023-07-12 09:00:00    190.771767
   2023-07-12 10:00:00    190.389116
   2023-07-12 11:00:00    189.314387
   2023-07-12 12:00:00    189.224929
   2023-07-12 13:00:00    189.812174
                              ...
   2023-07-18 11:00:00    193.254420
   2023-07-18 12:00:00    192.732713
   2023-07-18 13:00:00    192.905636
   2023-07-18 14:00:00    193.604533
   2023-07-18 15:00:00    193.984174
   Freq: H, Name: Adj Close, Length: 151, dtype: float64
   ```

Figure 2.23: Resampled Series with the mean value in the interval

7. Generate the open, high, low, and closing values for the resampled time interval:

   ```
   resampled.ohlc()
   ```

   Here's the output:

date	open	high	low	close
2023-07-12 09:00:00	189.910004	191.625000	189.570007	190.710007
2023-07-12 10:00:00	190.619995	190.914993	189.662003	189.669907
2023-07-12 11:00:00	189.716003	189.889999	188.489899	188.639893
2023-07-12 12:00:00	188.897293	189.759995	188.580002	189.360992
2023-07-12 13:00:00	189.429993	190.009995	189.379898	189.919998
...	...	...	...	...
2023-07-18 11:00:00	193.489899	193.637802	192.825302	193.199997
2023-07-18 12:00:00	193.130005	193.130005	192.434998	192.520004
2023-07-18 13:00:00	192.660004	193.160004	192.585007	192.985001
2023-07-18 14:00:00	192.985001	193.949905	192.985001	193.789902
2023-07-18 15:00:00	193.669907	194.229996	193.570007	193.729996

Figure 2.24: Resampled Series with the open, high, low, and closing values in the interval

How it works...

The **`resample`** method is used for time series data resampling and alters the frequency. By specifying a time frequency string, such as **`D`** for daily or **`M`** for monthly, we can decide on the new sampling rate. This method creates a **`Resampler`** object, upon which aggregation or transformation functions such as **`mean`** or **`sum`** can be applied. The resulting DataFrame or Series reflects the data that's consolidated or redistributed to the desired frequency. There's also a choice in how the boundaries of the bins are defined. The **`closed`** parameter can dictate which side of the bin interval is closed, either **`right`** or **`left`**. Moreover, the **`label`** parameter specifies which bin edge label to use for labeling the aggregation result.

The **`resample`** method has several useful arguments:

- **`rule`**: This is the offset string or object representing target conversion.
- **`axis`**: This specifies the axis to be resampled. By default, it's **`0`** (index).
- **`closed`**: This specifies which side of the bin interval is closed. The options are **`right`** or **`left`**.

- **`label`**: This specifies which bin edge label to label the bucket with. The options are **`right`** or **`left`**.
- **`convention`**: This is used when resampling period data (a time series with **`PeriodIndex`**) with the **`start`** or **`end`** option. The default is **`end`**.
- **`kind`**: This is used when upsampling from low to high frequency. The options are **`timestamp`** or **`period`**. By default, **`timestamp`** is used.
- **`loffset`**: This adjusts the resampled time labels.
- **`base`**: This is used for adjusting the start of the bins to a different timestamp.
- **`on`**: This is used for a DataFrame, to resample based on the time of a particular column rather than the DataFrame index.
- **`level`**: This is used for a **`MultiIndex`** DataFrame, to resample based on the time of a particular level of the **`MultiIndex`** object.
- **`origin`**: This defines the origin of the adjusted timestamps.
- **`offset`**: This adjusts the start of the bins based on this time delta.

There's more...

The **`resample`** and **`asfreq`** methods in pandas are both used to change the frequency of time series data, but they serve slightly different purposes and can be used in different scenarios. The **`resample`** method is primarily used for downsampling, where it provides different ways to aggregate the data for the new frequency (such as taking the mean, sum, maximum, minimum, and so on). When upsampling, **`resample`** can also interpolate the missing values by passing an arbitrary function to perform binning over a Series or DataFrame object in bins of arbitrary size.

The **`asfreq`** method is used when you want to convert a time series into a specified frequency. It does not provide any aggregation or transformation – it simply changes the frequency of the data. If you're upsampling (increasing the frequency), **`asfreq`** will introduce **`nan`** values for the newly created data points. If you're downsampling (decreasing the frequency), **`asfreq`** will drop the data points that don't fit into the new frequency.

pandas offers over 40 offsets that we can use with both **`resample`** and **`asfreq`**. They allow for even more flexibility than passing in **`D`** or **`H`** (for day or minute). You can see a full list by running **`pd.offsets.__all__`**.

Downsample the minute data to daily and adjust the labels in the index so that they match the frequency.

```
df.asfreq("D").to_period()
```

The result is a DataFrame. Note that the time portion of the index is dropped. Also, note the **`nan`** values on the days when there is no market data:

date	Open	High	Low	Close	Adj Close	Volume
2023-08-16	177.130005	177.639999	177.000000	177.494995	177.494995	3608898.0
2023-08-17	177.139999	177.505402	177.070007	177.339996	177.339996	2969302.0
2023-08-18	172.300003	173.089996	171.960007	172.945007	172.945007	4147338.0
2023-08-19	NaN	NaN	NaN	NaN	NaN	NaN
2023-08-20	NaN	NaN	NaN	NaN	NaN	NaN
2023-08-21	175.070007	175.119995	174.660004	174.731995	174.731995	1938994.0
2023-08-22	177.059998	177.085007	176.580002	176.602997	176.602997	1789325.0

Figure 2.25: DataFrame with frequency changed from 1 minute to calendar days

The **`asfreq`** method incorporates a **`method`** argument, directing pandas on how to handle **`nan`** values. This argument permits either **`backfill`** to populate **`nan`** values using the preceding valid observation or **`pad`** to populate with the subsequent valid observation. Additionally, the **`fill_value`** parameter lets us specify a custom value for filling in missing entries during the upsampling process.

If we want to avoid including days with no market prices, we can use the business day offset:

```
df.asfreq(pd.offsets.BDay())
```

The result is a DataFrame that only includes valid business days in the index:

date	Open	High	Low	Close	Adj Close	Volume
2023-08-16 09:30:00	177.130005	177.639999	177.000000	177.494995	177.494995	3608898
2023-08-17 09:30:00	177.139999	177.505402	177.070007	177.339996	177.339996	2969302
2023-08-18 09:30:00	172.300003	173.089996	171.960007	172.945007	172.945007	4147338
2023-08-21 09:30:00	175.070007	175.119995	174.660004	174.731995	174.731995	1938994
2023-08-22 09:30:00	177.059998	177.085007	176.580002	176.602997	176.602997	1789325

Figure 2.26: DataFrame with frequency changed from 1 minute to business days

> *IMPORTANT*
>
> **`resample`** *applies an aggregate to the data within the selected interval (for example,* **`ohlc`***), whereas* **`asfreq`** *changes the index and does not aggregate the data. This is apparent in Figure 2.23, where the time portion of the index is displayed. The values that match the given index (for example,* **`2023-07-12 09:30:00`***) are selected for inclusion in the returned DataFrame. We can see this by selecting data from the first row using* **`df.loc["2023-07-12 09:30:00"]`***. The result is a Series with values that match those in the first row.*

See also

See the documentation for more about the differences between **`resample`** and **`asfreq`**:

- Documentation on the pandas **`resample`** method: **https://pandas.pydata.org/docs/reference/api/pandas.DataFrame.resample.html**
- Documentation on the pandas **`asfreq`** method: **https://pandas.pydata.org/docs/reference/api/pandas.DataFrame.asfreq.html**

Addressing missing data issues

pandas is well-suited to handling missing data in time series data. In the context of financial market data, missing data can occur for various reasons:

- **Market closures**: Most financial markets aren't open 24/7. They operate on specific days and hours, closing for weekends, public holidays, or special events. If a data source tries to retrieve data when the market is closed, it might represent this as a missing value.
- **Data availability**: Not all historical data is available for every market or every security. Some markets may only have data available from a certain date, or some data may be missing due to technological issues, glitches, or errors during data recording and transmission.
- **Delisting of securities**: If a security gets delisted from a market (for example, a company going out of business), no new data is produced for that security. If your timeframe extends beyond the delisting date, missing data will be encountered.
- **Data granularity**: The level of detail in the dataset might also influence the existence of missing data. For example, if you're looking for minute-by-minute data but your source only provides hourly data, you'll have missing data for each minute that isn't the start of a new hour.

pandas has ways of handling missing data. This recipe presents those most common for algorithmic trading.

Getting ready...

We'll demonstrate several ways of filling in missing data. We'll start with AAPL's stock price data:

1. Import the necessary libraries:

```
import numpy as np
import pandas as pd
from openbb import obb
obb.user.preferences.output_type = "dataframe"
```

2. Download the stock price data:

```
df = obb.equity.price.historical(
    "AAPL",
    start_date="2020-07-01",
    end_date="2023-07-06",
    provider="yfinance",
)
```

3. **`df`** only contains the trading days for AAPL, so we'll reindex the DataFrame so that it includes all calendar days between the start and end date of the time series. First, create a **`DatetimeIndex`** object with calendar days:

```
calendar_dates = pd.date_range(
    start=df.index.min(),
    end=df.index.max(),
    freq="D"
)
```

4. Then, reindex the DataFrame:

```
calendar_prices = df.reindex(calendar_dates)
```

5. The resulting DataFrame is populated with **`nan`** values for dates where there is no data:

	Open	High	Low	Close	Adj Close	Volume	Dividends	Stock Splits
2020-07-01	89.498016	90.047081	89.201423	89.250443	89.250443	110737200.0	0.0	0.0
2020-07-02	90.167194	90.809406	89.135244	89.250443	89.250443	114041600.0	0.0	0.0
2020-07-03	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN
2020-07-04	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN
2020-07-05	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN
...	...	...	...	...	...	...	...	...
2023-07-02	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN
2023-07-03	193.518682	193.618553	191.501402	192.200470	192.200470	31458200.0	0.0	0.0
2023-07-04	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN
2023-07-05	191.311658	192.719744	190.362927	191.071976	191.071976	46920300.0	0.0	0.0
2023-07-06	189.583986	191.761054	188.944850	191.551331	191.551331	45094300.0	0.0	0.0

Figure 2.27: A DataFrame of stock prices with missing data on non-trading days

How to do it...

We'll demonstrate several examples of filling in the missing values:

1. Use the pandas **`fillna`** method with the **`method`** argument set to **`bfill`** to replace **`nan`** values with the next valid observation:

```
calendar_prices.bfill()
```

The result is a filled DataFrame:

	Open	High	Low	Close	Adj Close	Volume	Dividends	Stock Splits
2020-07-01	89.498016	90.047081	89.201423	89.250443	89.250443	110737200.0	0.0	0.0
2020-07-02	90.167194	90.809406	89.135244	89.250443	89.250443	114041600.0	0.0	0.0
2020-07-03	90.694182	92.110972	90.662316	91.637894	91.637894	118655600.0	0.0	0.0
2020-07-04	90.694182	92.110972	90.662316	91.637894	91.637894	118655600.0	0.0	0.0
2020-07-05	90.694182	92.110972	90.662316	91.637894	91.637894	118655600.0	0.0	0.0
...	...	...	...	...	...	...	...	...
2023-07-02	193.518682	193.618553	191.501402	192.200470	192.200470	31458200.0	0.0	0.0
2023-07-03	193.518682	193.618553	191.501402	192.200470	192.200470	31458200.0	0.0	0.0
2023-07-04	191.311658	192.719744	190.362927	191.071976	191.071976	46920300.0	0.0	0.0
2023-07-05	191.311658	192.719744	190.362927	191.071976	191.071976	46920300.0	0.0	0.0
2023-07-06	189.583986	191.761054	188.944850	191.551331	191.551331	45094300.0	0.0	0.0

Figure 2.28: A DataFrame with missing data backfilled to the previous NaN

2. Use **`ffill`** to propagate the last valid observation forward:

```
calendar_prices.ffill()
```

The result is a filled DataFrame:

	Open	High	Low	Close	Adj Close	Volume	Dividends	Stock Splits
2020-07-01	89.498016	90.047081	89.201423	89.250443	89.250443	110737200.0	0.0	0.0
2020-07-02	90.167194	90.809406	89.135244	89.250443	89.250443	114041600.0	0.0	0.0
2020-07-03	90.167194	90.809406	89.135244	89.250443	89.250443	114041600.0	0.0	0.0
2020-07-04	90.167194	90.809406	89.135244	89.250443	89.250443	114041600.0	0.0	0.0
2020-07-05	90.167194	90.809406	89.135244	89.250443	89.250443	114041600.0	0.0	0.0
...	...	...	...	...	...	...	...	...
2023-07-02	191.371579	194.217727	191.002068	193.708420	193.708420	85069600.0	0.0	0.0
2023-07-03	193.518682	193.618553	191.501402	192.200470	192.200470	31458200.0	0.0	0.0
2023-07-04	193.518682	193.618553	191.501402	192.200470	192.200470	31458200.0	0.0	0.0
2023-07-05	191.311658	192.719744	190.362927	191.071976	191.071976	46920300.0	0.0	0.0
2023-07-06	189.583986	191.761054	188.944850	191.551331	191.551331	45094300.0	0.0	0.0

Figure 2.29: A DataFrame with missing data backfilled to the next NaN

How it works...

The **`fillna`** method in pandas has several arguments that provide different ways to handle missing values:

- **`value`**: This specifies the value to use to fill NA/NaN values. It could be a scalar, dictionary, or Series.
- **`axis`**: This determines whether to fill missing values along rows (**`0`** or **`index`**) or columns (**`1`** or **`columns`**).
- **`inplace`**: If set to **`True`**, the operation modifies the data in place. The default is **`False`**.
- **`limit`**: Limits the number of consecutive forward/backward filled values.
- **`downcast`**: A dictionary of **`item->dtype`** that specifies the type of data for each item.

There's more...

Sometimes, simple forward-filling or back-filling techniques may not suffice, especially in more complex scenarios, such as generating implied volatility surfaces for derivatives pricing. In these cases, we need a more sophisticated method of dealing with missing data. The pandas **`interpolate`** method uses various interpolation techniques to fill missing values, including linear, polynomial, time, and spatial interpolations. The choice of interpolation method can be adapted based on the characteristics of the data, allowing for more accurate handling of missing values in complex structures such as implied volatility surfaces.

Here's an example of using linear interpolation to fill in the missing values in the DataFrame:

```
calendar_prices.interpolate(method="linear")
```

Here's an example of using cubic spline interpolation to fill in the missing values in the DataFrame:

```
calendar_prices.interpolate(method="cubicspline")
```

In both cases, the result is a DataFrame with filled values.

See also

For more about the `fillna` method and interpolation, take a look at the following documentation:

- Documentation on the pandas `fillna` method: https://pandas.pydata.org/docs/reference/api/pandas.DataFrame.fillna.html
- Documentation on the pandas `interpolate` method: https://pandas.pydata.org/docs/reference/api/pandas.DataFrame.interpolate.html
- More on interpolation: https://en.wikipedia.org/wiki/Interpolation

Applying custom functions to analyze time series data

Custom functions let us apply transformations and computations to data beyond the standard pandas methods. This flexibility becomes important when dealing with unique or complex analytical tasks. The `apply` function accepts a custom or built-in function as an argument and applies this function across the DataFrame or Series, returning an output containing the results.

> *IMPORTANT*
>
> *The `apply` function is notoriously slow since it operates on every row or column in a loop. Use this with caution when dealing with large DataFrames!*

Getting ready...

We assume that you've followed the instructions from the previous recipes and have a DataFrame called `df` with a `DatetimeIndex` object as the index.

How to do it...

We'll use the range as our custom function for this recipe:

1. Determine the range (high price minus low price) of the stock prices using an anonymous lambda function:

```
df.apply(lambda x: x["high"] - x["low"], axis=1)
```

2. Apply a user-defined function that does the same thing:

```
def fcn(row):
    return row["high"] - row["low"]
df.apply(fcn, axis=1)
```

The result in both cases is a Series with the high minus low price:

```
date
2020-07-01    0.846799
2020-07-02    1.676423
2020-07-06    1.450613
2020-07-07    1.568424
2020-07-08    1.261619
                ...
2023-06-29    1.130005
2023-06-30    3.220001
2023-07-03    2.120010
2023-07-05    2.360001
2023-07-06    2.807999
Length: 758, dtype: float64
```

Figure 2.30: Series with the range of AAPL stock

3. Test the validity of the data by flagging where the closing price is less than the low price or greater than the high price:

```
df["valid"] = df.apply(
    lambda x: x["low"] <= x["close"] <= x["high"], axis=1)
df[df.valid == False]
```

The result is an empty DataFrame since no values were invalid:

date	Open	High	Low	Close	Adj Close	Volume	Dividends	Stock Splits	range	valid

Figure 2.31: Empty DataFrame showing now records where the closing price is outside the high and low of the day

How it works...

The pandas **`apply`** function can accept an anonymous lambda function or a user-defined function. The code for computing the range of stock prices using a lambda is equivalent to the code using a user-defined function. We pass a 1 to the axis argument to apply the calculation along each row.

You can expand the use of the **`apply`** function with its other arguments:

- **`func`**: The function to apply to each row or column of the DataFrame
- **`axis`**: Whether to apply the function to each row (**`0`** or **`index`**) or column (**`1`** or **`columns`**)
- **`broadcast`**: Whether to broadcast the output of the function back onto the DataFrame
- **`raw`**: If **`True`**, the function receives arrays instead of Series objects or DataFrames
- **`reduce`**: If **`True`**, try to apply reduction procedures
- **`result_type`**: Controls the type of output: **`expand`**, **`reduce`**, **`broadcast`**, or **`None`**
- **`args`**: Additional positional arguments to pass to **`func`**

There's more...

The function that's passed to the **`apply`** method may accept additional arguments. Let's say we're interested in calculating the price range between two columns but we want to ignore the price ranges that are below a certain threshold. Our user-defined function needs to accept the two columns for which we compute the range and the threshold value.

First, we'll define the function that calculates the price range. This function will take four arguments: a row from the DataFrame, a **`high`** column, a **`low`** column, and a threshold for the minimum range:

```
def calculate_range(row, high_col, low_col, threshold):
    range = row[high_col] - row[low_col]
    return range if range > threshold else np.nan
```

IMPORTANT NOTE

> *When the pandas* **`apply`** *function is invoked with* **`axis=1`***, it applies the specified function to each row of the DataFrame. Each row is treated as a pandas Series object, with the index of the Series being the column names of the DataFrame. This Series object, representing the entire row, is passed as an argument to the function you've defined.*

This function computes the difference between the high and low prices. If this difference is greater than the threshold, it returns the difference; otherwise, it returns **`nan`**.

Next, we can use the **`apply`** method to apply this function to the DataFrame. The **`args`** parameter of **`apply`** allows us to pass extra arguments to the function:

```
threshold = 1.5
df["range"] = df.apply(
    calculate_range,
    args=("high", "low", threshold),
    axis=1
)
df.range
```

The result of this operation is a new column in the DataFrame that contains the computed price range if it's above the threshold, or **`nan`** otherwise:

```
date
2020-07-17         NaN
2020-07-20    2.437500
2020-07-21    2.507500
2020-07-22         NaN
2020-07-23    5.067497
                ...
2023-07-17    2.510010
2023-07-18    1.910004
2023-07-19    5.580002
2023-07-20    3.970001
2023-07-21    3.740005
Name: range, Length: 758, dtype: float64
```

Figure 2.32: Series with NaN values where the difference between the high and low price does not exceed a threshold

See also

See the following documentation for more on the **`apply`** method: **https://pandas.pydata.org/docs/reference/api/pandas.DataFrame.pct_change.html**.