Programming with Data: Advanced Python and Pandas

Time Series

Time Series

No new classes for Series and DataFrame.

A "timeseries" is a `Series` or `DataFrame` with a time index of some type.

Create a simple time series

In [2]:
```
ts = pd.Timestamp('2000-01-01 00:00')
```

In [3]:
```
s1 = pd.Series(101.7, index=[ts])
s1
```

Out[3]:
```
2000-01-01    101.7
dtype: float64
```

In [4]:
```
s1.index
```

Out[4]:
```
DatetimeIndex(['2000-01-01'], dtype='datetime64[ns]', freq=None)
```

Time Series Utility Functions

In [5]:
```
# Create a year's worth of business dates
dts = pd.date_range('2000-01-01', '2001-12-31', freq='B')
ts = pd.Series(np.random.randn(len(dts)), index=dts)
ts.head()
```

Out[5]:
```
2000-01-03   -0.011498
2000-01-04   -0.085123
2000-01-05    0.075910
2000-01-06   -1.649279
2000-01-07   -0.751812
Freq: B, dtype: float64
```

Selecting from a Time Series

In [6]:
```
ts['2000-03-20'] # ts.loc['2000-03-20']
```

Out[6]:
```
1.3091432326146675
```

In [7]:
```
ts['2000-03-24':'2000-03-30']
```

Out[7]:
```
2000-03-24    0.360544
2000-03-27   -0.820640
2000-03-28    0.106317
2000-03-29    1.364270
2000-03-30   -0.490908
Freq: B, dtype: float64
```

Select by month

In [8]:
```
ts['2000-02'].head()
```

Out[8]:
```
2000-02-01    1.157154
2000-02-02    2.183902
2000-02-03   -0.286083
2000-02-04    1.283730
2000-02-07    0.570529
Freq: B, dtype: float64
```

Select by year

In [9]:
```
ts['2000'].head()
```

Out[9]:
```
2000-01-03   -0.011498
2000-01-04   -0.085123
2000-01-05    0.075910
2000-01-06   -1.649279
2000-01-07   -0.751812
Freq: B, dtype: float64
```

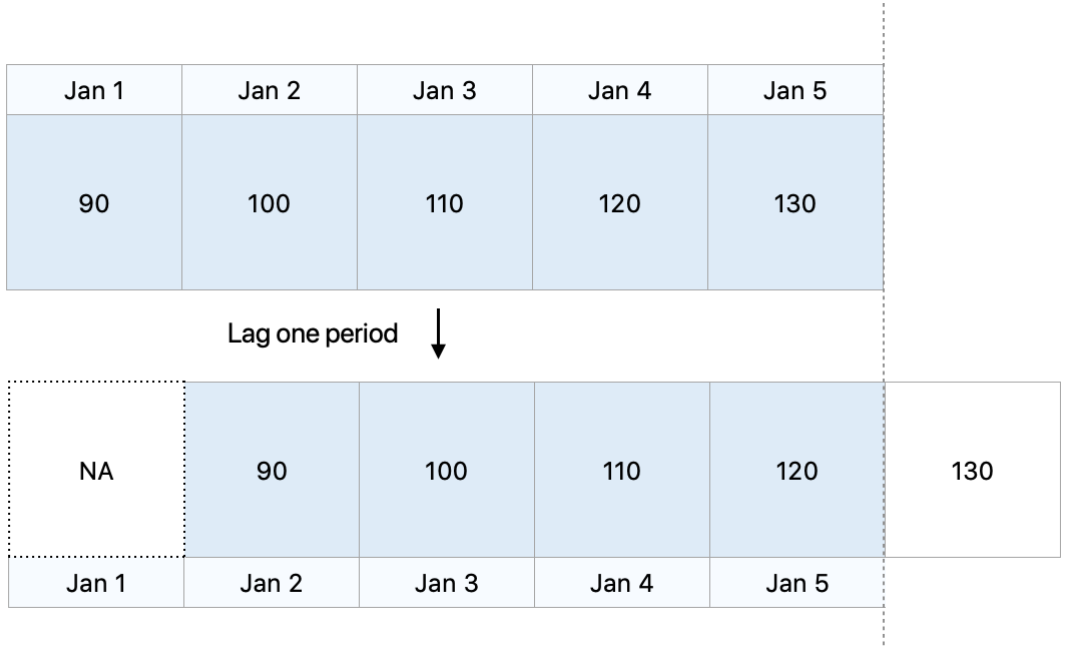
Leading and Lagging

- Common time series operation
- Useful for calculating things differences, percent change over time

Lagging

In [10]:
```
image('lag.png')
```

Out[10]:



Lagging using position

In [11]:
```
ts2k = ts['2000-01'].copy()
ts2k.shift(1).iloc[[0, 1, 2, -2, -1]] # show first and last few rows
```

Out[11]:
```
2000-01-03         NaN
2000-01-04   -0.011498
2000-01-05   -0.085123
2000-01-28   -1.724383
2000-01-31   -0.858197
dtype: float64
```

Notice that the `index` is constant. We drop the first value.

Lagging using index

The values remain constant but the index shifts

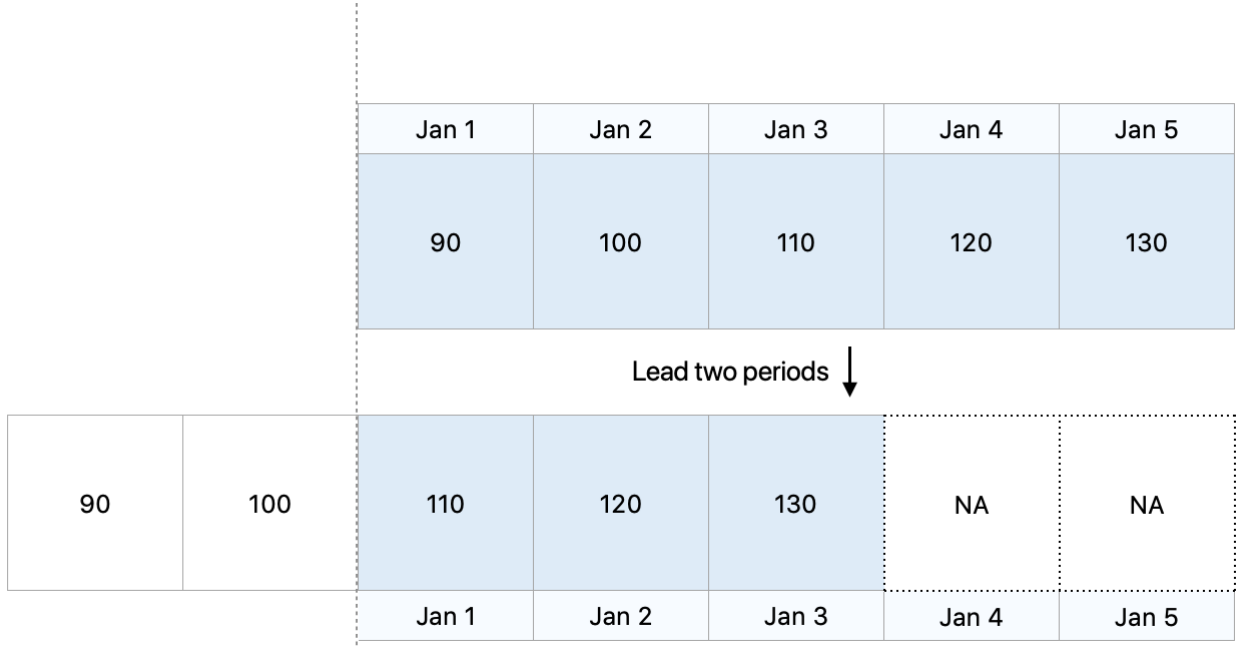
In [12]:
```
ts2k.tshift(1).iloc[[0, 1, 2, -2, -1]]
```

Out[12]:
```
2000-01-04   -0.011498
2000-01-05   -0.085123
2000-01-06    0.075910
2000-01-31   -0.858197
2000-02-01    1.681411
dtype: float64
```

Leading data

In [13]:
```
image('lead.png')
```

Out[13]:



Changing Frequencies: Resampling

Resampling is similar to grouping, expect with time and notions of forwards and backwards.

In [14]:
```
dts1 = pd.date_range('2000-01-01', '2000-03-31', freq='D')
ts3 = pd.Series(np.random.randn(len(dts1)), index=dts1)
```

In [15]:
```
grp = ts3.resample('M')
```

Resampling is like grouping

In [16]:
```
grp.mean()
```

Out[16]:
```
2000-01-31   -0.210334
2000-02-29   -0.019870
2000-03-31    0.213579
Freq: M, dtype: float64
```

In [17]:
```
grp.agg(['mean', 'std'])
```

Out[17]:

	mean	std
2000-01-31	-0.210334	0.894538
2000-02-29	-0.019870	1.051419
2000-03-31	0.213579	0.970263

Filling Missing Data

In [18]:
```
def make_series_4():
    dts = pd.date_range('2000-01-02', '2000-01-07', freq='D')
    rv = pd.Series(np.random.randn(len(dts)), index=dts)
    rv.iloc[[0, 2, 5]] = np.nan
    return rv

ts4 = make_series_4()
ts4
```

Out[18]:
```
2000-01-02         NaN
2000-01-03    1.301518
2000-01-04         NaN
2000-01-05   -1.452010
2000-01-06    1.001597
2000-01-07         NaN
Freq: D, dtype: float64
```

Filling Data Forward

In [19]:
```
ts4.ffill()
```

Out[19]:
```
2000-01-02         NaN
2000-01-03    1.301518
2000-01-04    1.301518
2000-01-05   -1.452010
2000-01-06    1.001597
2000-01-07    1.001597
Freq: D, dtype: float64
```

Filling Data Backwards

In [20]:
```
ts4.bfill()
```

Out[20]:
```
2000-01-02    1.301518
2000-01-03    1.301518
2000-01-04   -1.452010
2000-01-05   -1.452010
2000-01-06    1.001597
2000-01-07         NaN
Freq: D, dtype: float64
```

Filling with limits

In [21]:
```
def make_series_5():
    dts = pd.date_range('2000-01-01', '2000-01-05', freq='D')
    rv = pd.Series(np.random.randn(len(dts)), index=dts)
    rv[1:] = np.nan
    return rv

ts5 = make_series_5()
ts5
```

Out[21]:
```
2000-01-01    1.33151
2000-01-02        NaN
2000-01-03        NaN
2000-01-04        NaN
2000-01-05        NaN
Freq: D, dtype: float64
```

Filling with limits (cont)

In [22]:
```
ts5.ffill(limit=2)
```

Out[22]:
```
2000-01-01    1.33151
2000-01-02    1.33151
2000-01-03    1.33151
2000-01-04        NaN
2000-01-05        NaN
Freq: D, dtype: float64
```

Aligning Dates

Sample Data

Create a weekly and monthly series.

In [23]:
```
dts_m = pd.bdate_range('2000-01', periods=2, freq='MS')
t_bill = pd.Series([0.012, 0.023], index=dts_m)

dts_d = pd.bdate_range('2000-01-01', periods=8, freq='W')
sp5_weekly = random_series(dts_d)
```

View the data

In [24]:
```
sp5_weekly
```

Out[24]:
```
2000-01-02   -1.352033
2000-01-09   -0.185830
2000-01-16   -1.156032
2000-01-23   -0.608064
2000-01-30   -1.120766
2000-02-06   -2.069586
2000-02-13   -0.499210
2000-02-20   -0.201274
Freq: W-SUN, dtype: float64
```

In [25]:
```
t_bill
```

Out[25]:
```
2000-01-01    0.012
2000-02-01    0.023
Freq: MS, dtype: float64
```

Using `reindex`

In [26]:
```
t_bill.reindex(sp5_weekly.index)
```

Out[26]:
```
2000-01-02   NaN
2000-01-09   NaN
2000-01-16   NaN
2000-01-23   NaN
2000-01-30   NaN
2000-02-06   NaN
2000-02-13   NaN
2000-02-20   NaN
Freq: W-SUN, dtype: float64
```

Reindex and fill forward

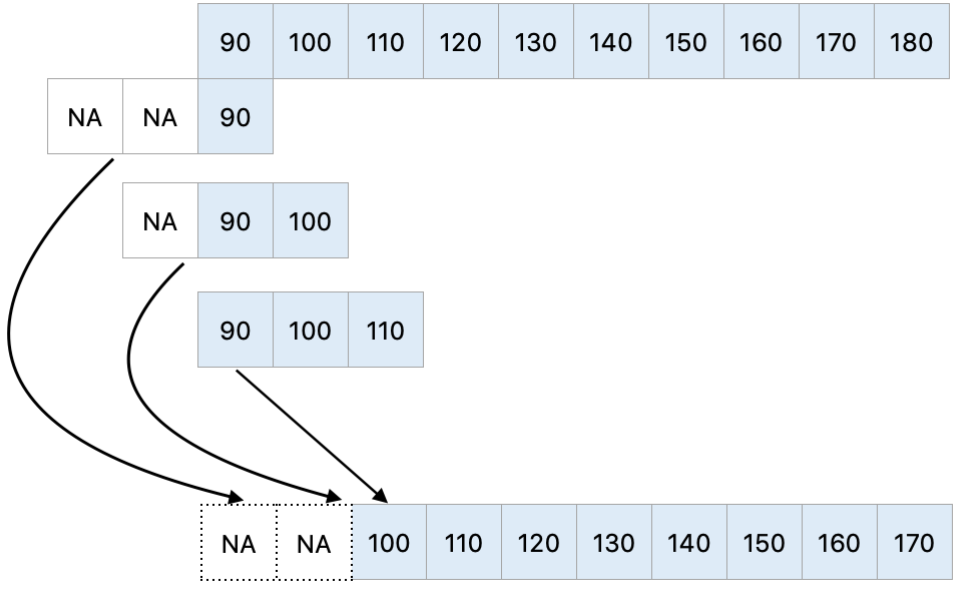
In [27]:
```
t_bill.reindex(sp5_weekly.index, method='ffill')
```

Out[27]:
```
2000-01-02    0.012
2000-01-09    0.012
2000-01-16    0.012
2000-01-23    0.012
2000-01-30    0.012
2000-02-06    0.023
2000-02-13    0.023
2000-02-20    0.023
Freq: W-SUN, dtype: float64
```

Rolling Calculations

In [28]:
```
image('rolling.png')
```

Out[28]:



Rolling example

In [29]:
```
roll = sp5_weekly.rolling(2)
```

In [30]:
```
roll.mean()
```

Out[30]:
```
2000-01-02         NaN
2000-01-09   -0.768931
2000-01-16   -0.670931
2000-01-23   -0.882048
2000-01-30   -0.864415
2000-02-06   -1.595176
2000-02-13   -1.284398
2000-02-20   -0.350242
Freq: W-SUN, dtype: float64
```

Programming with Data: Advanced Python and Pandas

Merge, Join, & Combine

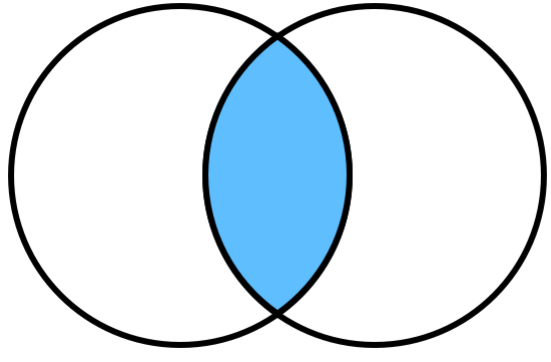
Types of Joins

- Inner
- Left
- Right
- Full

Inner Join

In [2]:
```
Image(filename='assets/inner-join.png', retina=True)
```

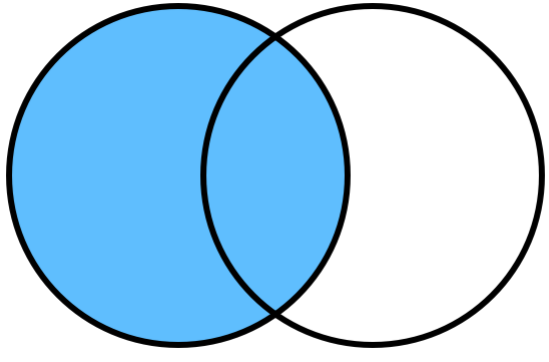
Out[2]:



Left/Right Join

In [3]:
```
Image(filename='assets/left-join.png', retina=True)
```

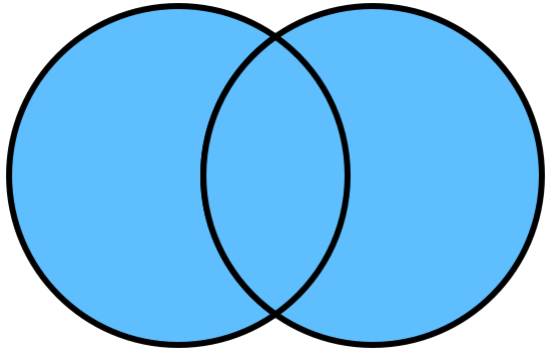
Out[3]:



Full Join

In [4]:
```
Image(filename='assets/full-join.png', retina=True)
```

Out[4]:



The Data

Photo by Chris Liverani on Unsplash

Somewhat simple data

In [5]:
```
df1 = pd.DataFrame({
    'ticker': ['AAPL', 'MSFT', 'IBM', 'YHOO', 'GOOG'],
    'open': [426.23, 42.30, 101.65, 35.53, 200.41]
})
df1
```

Out[5]:

	ticker	open
0	AAPL	426.23
1	MSFT	42.30
2	IBM	101.65
3	YHOO	35.53
4	GOOG	200.41

More somewhat simple data

Tickers and close prices. Additional ticker for `NFLX`.

In [6]:
```
df2 = pd.DataFrame({
    'ticker': ['AAPL', 'GOOG', 'NFLX'],
    'close': [427.53, 210.96, 91.86]
}, columns=['ticker', 'close'])
df2
```

Out[6]:

	ticker	close
0	AAPL	427.53
1	GOOG	210.96
2	NFLX	91.86

Coding an inner join

An inner join gives us the intersection of the keys.

In [7]:
```
df1m2 = pd.merge(df1, df2, on='ticker')
df1m2
```

Out[7]:

	ticker	open	close
0	AAPL	426.23	427.53
1	GOOG	200.41	210.96

Verifying the inner join

We drop everything except tickers that are present **both** data frames.

In [8]:
```
common_tickers = set(df1.ticker) & set(df2.ticker)
common_tickers
```

Out[8]:
```
{'AAPL', 'GOOG'}
```

In [9]:
```
assert set(df1m2.ticker) == common_tickers
```

Aside: 99% of the time, use `pd.merge`

Most flexible way to join two data frames

- `pd.concat` is more general - useful to join a collection (e.g. `list`) of data frames
- `pd.DataFrame.join` works in more specific circumstances

Left Join

Include all keys from the `left` data frame.

In [10]:
```
df1m2_left = pd.merge(df1, df2, on='ticker', how='left')
df1m2_left
```

Out[10]:

	ticker	open	close
0	AAPL	426.23	427.53
1	MSFT	42.30	NaN
2	IBM	101.65	NaN
3	YHOO	35.53	NaN
4	GOOG	200.41	210.96

In [11]:
```
assert set(df1.ticker) == set(df1m2_left.ticker)
```

Filling missing levels

In [12]:
```
df1m2_left
```

Out[12]:

	ticker	open	close
0	AAPL	426.23	427.53
1	MSFT	42.30	NaN
2	IBM	101.65	NaN
3	YHOO	35.53	NaN
4	GOOG	200.41	210.96

Notice that `pandas` fills missing levels from `df2` with `NaN`. Comparable to `SQL` where values would be `NULL`.

Right Join

Include all keys from the `right` data frame.

In [13]:
```
pd.merge(df1, df2, on='ticker', how='right')
```

Out[13]:

	ticker	open	close
0	AAPL	426.23	427.53
1	GOOG	200.41	210.96
2	NFLX	NaN	91.86

Same missingness handling as `left` join.

Outer/Full Join

In [14]:
```
df1m2_full = pd.merge(df1, df2, on='ticker', how='outer')
df1m2_full
```

Out[14]:

	ticker	open	close
0	AAPL	426.23	427.53
1	MSFT	42.30	NaN
2	IBM	101.65	NaN
3	YHOO	35.53	NaN
4	GOOG	200.41	210.96
5	NFLX	NaN	91.86

In [15]:
```
assert set(df1.ticker) | set(df2.ticker) == set(df1m2_full.ticker)
```

Concatenation/Binding

- Add rows* (`append`)
- Add columns (`pd.concat`)
- Add rows and columns (`pd.concat` or `append`)

* `append` will also add columns!

Data for Concatenation

In [16]:
```
df3 = df1.assign(date=pd.datetime(2018, 1, 4))\
    .iloc[:2, ] # first 2 rows only
df3
```

```
/Users/daniel/miniconda3/envs/progwd/lib/python3.7/site-packages/ipykernel_launcher.py:1: FutureWarning: The pandas.datetime class is deprecated and will be removed from pandas in a future version. Import from datetime module instead.
  """Entry point for launching an IPython kernel.
```

Out[16]:

	ticker	open	date
0	AAPL	426.23	2018-01-04
1	MSFT	42.30	2018-01-04

In [17]:
```
df4 = df3.assign(
    date=pd.datetime(2018, 1, 5),
    open=lambda x: x.open + 10
)
df4
```

```
/Users/daniel/miniconda3/envs/progwd/lib/python3.7/site-packages/ipykernel_launcher.py:2: FutureWarning: The pandas.datetime class is deprecated and will be removed from pandas in a future version. Import from datetime module instead.
```

Out[17]:

	ticker	open	date
0	AAPL	436.23	2018-01-05
1	MSFT	52.30	2018-01-05

Adding rows

In [18]:
```
df3.append(df4)
```

Out[18]:

	ticker	open	date
0	AAPL	426.23	2018-01-04
1	MSFT	42.30	2018-01-04
0	AAPL	436.23	2018-01-05
1	MSFT	52.30	2018-01-05

Notice how the index is repeated and duplicated for the default `pd.RangeIndex`

No dups please

To check for duplicated index values:

In [19]:
```
try:
    df3.append(df4, verify_integrity=True)
except ValueError as e:
    print(e)
```

```
Indexes have overlapping values: Int64Index([0, 1], dtype='int64')
```

Ignore the index

`ignore_index` discards the indexes from the bound data frames

In [20]:
```
df3.append(df4, ignore_index=True)
```

Out[20]:

	ticker	open	date
0	AAPL	426.23	2018-01-04
1	MSFT	42.30	2018-01-04
2	AAPL	436.23	2018-01-05
3	MSFT	52.30	2018-01-05

We usually don't need to validate the index when we pass `ignore_index` because we're creating a new index!

Rows and Columns with `append`

- `append` does an outer join on both rows and columns
- We'll see how to avoid this with `concat`

In [21]:
```
df3a = df3.assign(close=lambda x: (x.open + 9))
df3a
```

Out[21]:

	ticker	open	date	close
0	AAPL	426.23	2018-01-04	435.23
1	MSFT	42.30	2018-01-04	51.30

We've been warned

In [22]:
```
df3a.append(df4, ignore_index=True)
```

Out[22]:

	ticker	open	date	close
0	AAPL	426.23	2018-01-04	435.23
1	MSFT	42.30	2018-01-04	51.30
2	AAPL	436.23	2018-01-05	NaN
3	MSFT	52.30	2018-01-05	NaN

Probably the most common `append`

In [23]:
```
df3a.append(df4, ignore_index=True, sort=False)
```

Out[23]:

	ticker	open	date	close
0	AAPL	426.23	2018-01-04	435.23
1	MSFT	42.30	2018-01-04	51.30
2	AAPL	436.23	2018-01-05	NaN
3	MSFT	52.30	2018-01-05	NaN

Anytime you're repeating code, put it in a function and stay `DRY` (Don't repeat yourself)

In [24]:
```
def pwd_append(
    x: pd.DataFrame,
    y: pd.DataFrame,
    ignore_index=True,
    sort=False,
    verify_integrity=False
) -> pd.DataFrame:
    kwargs = {
        "ignore_index": ignore_index,
        "sort": sort,
        "verify_integrity": verify_integrity
    }
    return x.append(y, **kwargs)
```

General-purpose `pd.concat`

- **Join** and bind across rows or columns
- Pass 1 or more `Series` or `DataFrames`

Replicate `append`

- IMPORTANT: `ignore_index` only applies to the axis of concatenation which can be rows or columns

In [25]:
```
pd.concat([df3, df4], ignore_index=True)
```

Out[25]:

	ticker	open	date
0	AAPL	426.23	2018-01-04
1	MSFT	42.30	2018-01-04
2	AAPL	436.23	2018-01-05
3	MSFT	52.30	2018-01-05

Replicate `append` (cont)

Outer join of both rows and columns like `append`

In [26]:
```
pd.concat([df3a, df4], ignore_index=True, sort=False)
```

Out[26]:

	ticker	open	date	close
0	AAPL	426.23	2018-01-04	435.23
1	MSFT	42.30	2018-01-04	51.30
2	AAPL	436.23	2018-01-05	NaN
3	MSFT	52.30	2018-01-05	NaN

Stuff you can't do with `append`

Bind columns only

In [27]:
```
df5 = pd.DataFrame({'a': [1, 2]})
df6 = pd.DataFrame({'b': [3, 4]})
```

In [28]:
```
pd.concat([df5, df6], axis=1)
```

Out[28]:

	a	b
0	1	3
1	2	4

`concat` binds rows and columns

- Always performs an outer join on the concatenation axis

In [29]:
```
df6a = df6.set_index(pd.Index([6, 7]))
pd.concat([df5, df6a], sort=False)
```

Out[29]:

	a	b
0	1.0	NaN
1	2.0	NaN
6	NaN	3.0
7	NaN	4.0

Specify behavior of non-concatenation axis

- The `join` parameter only applies to the non-concatenation axis
- Set to `inner` to only get the common index elements or columns

In [30]:
```
pd.concat([df3a, df4], ignore_index=True, sort=False, join='inner')
```

Out[30]:

	ticker	open	date
0	AAPL	426.23	2018-01-04
1	MSFT	42.30	2018-01-04
2	AAPL	436.23	2018-01-05
3	MSFT	52.30	2018-01-05

Notice there is no `close` column because it's not present in both data frames

Identify the source Series/DataFrame with `keys`

In [31]:
```
pd.concat([df3, df4], keys=['df3', 'df4'])
```

Out[31]:

		ticker	open	date
df3	0	AAPL	426.23	2018-01-04
	1	MSFT	42.30	2018-01-04
df4	0	AAPL	436.23	2018-01-05
	1	MSFT	52.30	2018-01-05

Use `keys` and `names`

In [32]:
```
pd.concat([df3, df4], keys=['df3', 'df4'], names=['source', 'row_num'])
```

Out[32]:

		ticker	open	date
source	**row_num**			
df3	**0**	AAPL	426.23	2018-01-04
	1	MSFT	42.30	2018-01-04
df4	**0**	AAPL	436.23	2018-01-05
	1	MSFT	52.30	2018-01-05

Programming with Data: Advanced Python and Pandas

Advanced Merging & Reshaping

Grouped and Ordered Data

Working again with securities market data. In quant finance, this is a common data type, daily stock prices.

Display the data

In [4]:
```
_dts = ['2015-12-29', '2015-12-30', '2015-12-31', '2016-01-04']
_goog = [776.60, 771.00, 758.88, 741.84]
_aapl = [108.74, 107.32, 105.26, 105.35]

df = pd.DataFrame({
    'ticker': ['GOOG'] * 4 + ['AAPL'] * 4,
    'date': [pd.to_datetime(x) for x in _dts] * 2,
    'close': _goog + _aapl
})
df
```

Out[4]:

	ticker	date	close
0	GOOG	2015-12-29	776.60
1	GOOG	2015-12-30	771.00
2	GOOG	2015-12-31	758.88
3	GOOG	2016-01-04	741.84
4	AAPL	2015-12-29	108.74
5	AAPL	2015-12-30	107.32
6	AAPL	2015-12-31	105.26
7	AAPL	2016-01-04	105.35

A single, ordered series

In [5]:
```
tbill = pd.DataFrame({
    'date': [pd.to_datetime(x) for x in ['2015-12-30', '2016-01-04']],
    'rate': [2.40, 2.56]
})
tbill
```

Out[5]:

	date	rate
0	2015-12-30	2.40
1	2016-01-04	2.56

Merge data that is grouped and ordered

- Left panel is irregularly spaced, e.g. business days
- Right time series also irregularly spaced, e.g. a sparse subset of the first series

How not to do the merge

Don't use plain `pd.merge` and fill forward across groups.

In [6]:
```
pd.merge(df, tbill, on='date', how='left').ffill()
```

Out[6]:

	ticker	date	close	rate
0	GOOG	2015-12-29	776.60	NaN
1	GOOG	2015-12-30	771.00	2.40
2	GOOG	2015-12-31	758.88	2.40
3	GOOG	2016-01-04	741.84	2.56
4	AAPL	2015-12-29	108.74	2.56
5	AAPL	2015-12-30	107.32	2.40
6	AAPL	2015-12-31	105.26	2.40
7	AAPL	2016-01-04	105.35	2.56

Merge Ordered V2

In [7]:
```
mkt = pd.merge_ordered(df, tbill, on='date', left_by='ticker', fill_method='ffill'
)
mkt
```

Out[7]:

	ticker	date	close	rate
0	GOOG	2015-12-29	776.60	NaN
1	GOOG	2015-12-30	771.00	2.40
2	GOOG	2015-12-31	758.88	2.40
3	GOOG	2016-01-04	741.84	2.56
4	AAPL	2015-12-29	108.74	NaN
5	AAPL	2015-12-30	107.32	2.40
6	AAPL	2015-12-31	105.26	2.40
7	AAPL	2016-01-04	105.35	2.56

Reshaping & Pivoting

Wide and Long Formats

- Depending on the operation or the data storage location, data stored in a "wide" or "long" format

Long Format

- Common format for data in relational databases because allows new attributes without a schema change
- "Long" format is also called "stacked" or "record" format in the `pandas` documentation. Also called `Entity-Attribute-Value (EAV)`
- "Sparse" by design

Simplest Long Format

- Multiple attributes for a single entity (AAPL)
- Row for every period (12/29 & 12/30) x (number of attributes)

In [9]:
```
aapl_long = make_long_aapl()
aapl_long
```

Out[9]:

	date	ticker	variable	value
0	2015-12-29	AAPL	open	106.96
1	2015-12-29	AAPL	close	108.74
2	2015-12-30	AAPL	open	108.58
3	2015-12-30	AAPL	close	107.32

Wide Format

- Identifiers stored in the index
- Each attribute has its own column
- Common format for use by machine learning algorithms

Long-to-Wide

In [10]:
```
aapl_long
```

Out[10]:

	date	ticker	variable	value
0	2015-12-29	AAPL	open	106.96
1	2015-12-29	AAPL	close	108.74
2	2015-12-30	AAPL	open	108.58
3	2015-12-30	AAPL	close	107.32

In [11]:
```
aapl_long.pivot(index='date', columns='variable', values='value')
```

Out[11]:

variable	close	open
date		
2015-12-29	108.74	106.96
2015-12-30	107.32	108.58

Long-to-Wide with multiple ID columns

In [12]:
```
aapl_wide = aapl_long.set_index(['date', 'ticker', 'variable']).unstack()
aapl_wide
```

Out[12]:

		value	
	variable	close	open
date	ticker		
2015-12-29	**AAPL**	108.74	106.96
2015-12-30	**AAPL**	107.32	108.58

Wide-to-Long

In [13]:
```
aapl_wide.stack().reset_index()
```

Out[13]:

	date	ticker	variable	value
0	2015-12-29	AAPL	close	108.74
1	2015-12-29	AAPL	open	106.96
2	2015-12-30	AAPL	close	107.32
3	2015-12-30	AAPL	open	108.58

Pivot Tables

In [14]:
```
mkt
```

Out[14]:

	ticker	date	close	rate
0	GOOG	2015-12-29	776.60	NaN
1	GOOG	2015-12-30	771.00	2.40
2	GOOG	2015-12-31	758.88	2.40
3	GOOG	2016-01-04	741.84	2.56
4	AAPL	2015-12-29	108.74	NaN
5	AAPL	2015-12-30	107.32	2.40
6	AAPL	2015-12-31	105.26	2.40
7	AAPL	2016-01-04	105.35	2.56

Simple Pivot Table

In [15]:
```
pd.pivot_table(mkt, index='ticker', aggfunc='mean')
```

Out[15]:

	close	rate
ticker		
AAPL	106.67	2.45
GOOG	762.08	2.45