DSC 201: Data Analysis & Visualization

Time Series

Dr. David Koop
**Transform and Unwrapping**

```python
def normalize(x):
    return (x - x.mean()) / x.std()
```

```
In [84]: g.transform(normalize)
Out[84]:
0   -1.161895
1   -1.161895
2   -1.161895
3   -0.387298
4   -0.387298
5   -0.387298
6    0.387298
7    0.387298
8    0.387298
9    1.161895
10   1.161895
11   1.161895
Name: value, dtype: float64
```

```
In [85]: g.apply(normalize)
Out[85]:
0   -1.161895
1   -1.161895
2   -1.161895
3   -0.387298
4   -0.387298
5   -0.387298
6    0.387298
7    0.387298
8    0.387298
9    1.161895
10   1.161895
11   1.161895
Name: value, dtype: float64
```
Transform and Unwrapping

```python
def normalize(x):
    return (x - x.mean()) / x.std()
```

In [84]: g.transform(normalize)
Out[84]:
0   -1.161895
1    -1.161895
2    -1.161895
3   -0.387298
4   -0.387298
5   -0.387298
6    0.387298
7    0.387298
8    0.387298
9    1.161895
10   1.161895
11   1.161895
Name: value, dtype: float64

In [85]: g.apply(normalize)
Out[85]:
0   -1.161895
1    -1.161895
2    -1.161895
3   -0.387298
4   -0.387298
5   -0.387298
6    0.387298
7    0.387298
8    0.387298
9    1.161895
10   1.161895
11   1.161895
Name: value, dtype: float64

In [87]: normalized = (df['value'] - g.transform('mean')) / g.transform('std')

Fastest: "Unwrapped" group operation
## Pivot Tables: Split Indices and Columns

- `tips.pivot_table(['size'], index=['sex', 'day'], columns='smoker', aggfunc='sum', margins=True)`

<table>
<thead>
<tr>
<th></th>
<th>size</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>smoker</td>
</tr>
<tr>
<td>sex</td>
<td>day</td>
</tr>
<tr>
<td>Female</td>
<td>Fri</td>
</tr>
<tr>
<td></td>
<td>Sat</td>
</tr>
<tr>
<td></td>
<td>Sun</td>
</tr>
<tr>
<td></td>
<td>Thur</td>
</tr>
<tr>
<td>Male</td>
<td>Fri</td>
</tr>
<tr>
<td></td>
<td>Sat</td>
</tr>
<tr>
<td></td>
<td>Sun</td>
</tr>
<tr>
<td></td>
<td>Thur</td>
</tr>
<tr>
<td>All</td>
<td></td>
</tr>
</tbody>
</table>
Crosstabs: Pivot Tables with Counts

- `pd.crosstab([tips.time, tips.day], tips.smoker, margins=True)`

Or...

`tips.pivot_table('total_bill', index=['time', 'day'], columns=['smoker'], aggfunc='count', margins=True, fill_value=0)`

<table>
<thead>
<tr>
<th>smoker</th>
<th>No</th>
<th>Yes</th>
<th>All</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>time</strong></td>
<td><strong>day</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Dinner</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Fri</td>
<td>3</td>
<td>9</td>
<td>12</td>
</tr>
<tr>
<td>Sat</td>
<td>45</td>
<td>42</td>
<td>87</td>
</tr>
<tr>
<td>Sun</td>
<td>57</td>
<td>19</td>
<td>76</td>
</tr>
<tr>
<td>Thur</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>Lunch</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Fri</td>
<td>1</td>
<td>6</td>
<td>7</td>
</tr>
<tr>
<td>Thur</td>
<td>44</td>
<td>17</td>
<td>61</td>
</tr>
<tr>
<td><strong>All</strong></td>
<td>151</td>
<td>93</td>
<td>244</td>
</tr>
</tbody>
</table>
Types of Time Data

• Timestamps: specific instants in time (e.g. 2018-11-27 14:15:00)
• Periods: have a standard start and length (e.g. the month November 2018)
• Intervals: have a start and end timestamp
  - Periods are special case
  - Example: 2018-11-21 14:15:00 — 2018-12-01 05:15:00
• Elapsed time: measure of time relative to a start time (15 minutes)
Python Support for Time

- The `datetime` package
  - Has date, time, and datetime classes
  - `.now()` method: the current datetime
  - Can access properties of the time (year, month, seconds, etc.)

- Converting from strings to datetimes:
  - `datetime.strptime`: good for known formats
  - `dateutil.parser.parse`: good for unknown formats

- Converting to strings
  - `str(dt) or dt.strftime(<format>)`

- Differences between times
  - `datetime.timedelta`: can get number of days/hours/etc.
  - deal with issues with different length months, etc.
Pandas Support for Datetime

- **pd.to_datetime:**
  - convenience method
  - can convert an entire column to datetime
- Has a NaT to indicate a missing time value
- Stores in a numpy.datetime64 format
- **pd.Timestamp:** a wrapper for the datetime64 objects
- Accessing a particular time or checking equivalence allows any string that can be interpreted as a date:
  - ts['1/10/2011'] or ts['20110110']
- Use .dt accessor (like .str) to do datetime-specific operations
- Date ranges: pd.date_range('4/1/2012','6/1/2012', freq='4h')
Can use time as an **index**

```python
data = [('2017-11-30', 48),
       ('2017-12-02', 45),
       ('2017-12-03', 44),
       ('2017-12-04', 48)]
dates, temps = zip(*data)
s = pd.Series(temps, pd.to_datetime(dates))
```

Accessing a particular time or checking equivalence allows any string that can be interpreted as a date:

- `s['12/04/2017']` or `s['20171204']`

Using a less specific string will get all matching data:

- `s['2017-12']` returns the three December entries

Time slices do not need to exist:

- `s['2017-12-01':'2017-12-31']`
Shifting Time Series

• Data:

\[
[(\text{2017-11-30}', 48), (\text{2017-12-02}', 45),
(\text{2017-12-03}', 44), (\text{2017-12-04}', 48)]
\]

• Compute day-to-day difference in high temperature:

\[-s - s.shift(1) \text{ (same as } s.diff())\]

<table>
<thead>
<tr>
<th>Date</th>
<th>Difference</th>
</tr>
</thead>
<tbody>
<tr>
<td>2017-11-30</td>
<td>NaN</td>
</tr>
<tr>
<td>2017-12-02</td>
<td>-3.0</td>
</tr>
<tr>
<td>2017-12-03</td>
<td>-1.0</td>
</tr>
<tr>
<td>2017-12-04</td>
<td>4.0</td>
</tr>
</tbody>
</table>

\[-s - s.shift(1, 'd')\]

<table>
<thead>
<tr>
<th>Date</th>
<th>Difference</th>
</tr>
</thead>
<tbody>
<tr>
<td>2017-11-30</td>
<td>NaN</td>
</tr>
<tr>
<td>2017-12-01</td>
<td>NaN</td>
</tr>
<tr>
<td>2017-12-02</td>
<td>NaN</td>
</tr>
<tr>
<td>2017-12-03</td>
<td>-1.0</td>
</tr>
<tr>
<td>2017-12-04</td>
<td>4.0</td>
</tr>
<tr>
<td>2017-12-05</td>
<td>NaN</td>
</tr>
</tbody>
</table>
Timedelta

- Compute differences between dates
- Lives in `datetime` module
  
  ```python
  diff = parse_date("1 Jan 2017") - datetime.now().date()
  diff.days
  ```
- Also a `pd.Timedelta` object that takes strings:
  
  ```python
  datetime.now().date() + pd.Timedelta("4 days")
  ```
- Also, roll dates using anchored offsets
  
  ```python
  from pandas.tseries.offsets import Day, MonthEnd
  now = datetime(2011, 11, 17)
  In [107]: now + MonthEnd(2)
  Out[107]: Timestamp('2011-12-31 00:00:00')
  ```
Assignment 5

- Aggregation, Time Series, and Visualization
- Compare Hurricane Joaquin and Hurricane Maria
Time Zones

• Why?
• Coordinated Universal Time (UTC) is the standard time (basically equivalent to Greenwich Mean Time (GMT))
• Other time zones are UTC +/- a number in [1,12]
• Dartmouth is UTC-5 (aka US/Eastern)
Python, Pandas, and Time Zones

- Time series in pandas are **time zone native**
- The pytz module keeps track of all of the time zone parameters
  - even Daylight Savings Time
- Localize a timestamp using `tz_localize`
  - `ts = pd.Timestamp("1 Dec 2016 12:30 PM")`
  - `ts = ts.tz_localize("US/Eastern")`
- Convert a timestamp using `tz_convert`
  - `ts.tz_convert("Europe/Budapest")`
- Operations involving timestamps from different time zones become UTC
Frequency

- Generic time series in pandas are **irregular**
  - there is no fixed frequency
  - we don't necessarily have data for every day/hour/etc.

- Date ranges have frequency

```python
In [76]: pd.date_range(start='2012-04-01', periods=20)
Out[76]:
```
Lots of Frequencies (not comprehensive)

<table>
<thead>
<tr>
<th>Alias</th>
<th>Offset type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>D</td>
<td>Day</td>
<td>Calendar daily</td>
</tr>
<tr>
<td>B</td>
<td>BusinessDay</td>
<td>Business daily</td>
</tr>
<tr>
<td>H</td>
<td>Hour</td>
<td>Hourly</td>
</tr>
<tr>
<td>T or min</td>
<td>Minute</td>
<td>Minutely</td>
</tr>
<tr>
<td>S</td>
<td>Second</td>
<td>Secondly</td>
</tr>
<tr>
<td>L or ms</td>
<td>Milli</td>
<td>Millisecond (1/1,000 of 1 second)</td>
</tr>
<tr>
<td>U</td>
<td>Micro</td>
<td>Microsecond (1/1,000,000 of 1 second)</td>
</tr>
<tr>
<td>M</td>
<td>MonthEnd</td>
<td>Last calendar day of month</td>
</tr>
<tr>
<td>BM</td>
<td>BusinessMonthEnd</td>
<td>Last business day (weekday) of month</td>
</tr>
<tr>
<td>MS</td>
<td>MonthBegin</td>
<td>First calendar day of month</td>
</tr>
<tr>
<td>BMS</td>
<td>BusinessMonthBegin</td>
<td>First weekday of month</td>
</tr>
<tr>
<td>W-MON, W-TUE, ...</td>
<td>Week</td>
<td>Weekly on given day of week (MON, TUE, WED, THU, FRI, SAT, or SUN)</td>
</tr>
<tr>
<td>WOM-1MON, WOM-2MON, ...</td>
<td>WeekOfMonth</td>
<td>Generate weekly dates in the first, second, third, or fourth week of the month (e.g., WOM-3FRI for the third Friday of each month)</td>
</tr>
<tr>
<td>Q-JAN, Q-FEB, ...</td>
<td>QuarterEnd</td>
<td>Quarterly dates anchored on last calendar day of each month, for year ending in indicated month (JAN, FEB, MAR, APR, MAY, JUN, JUL, AUG, SEP, OCT, NOV, or DEC)</td>
</tr>
<tr>
<td>BQ-JAN, BQ-FEB, ...</td>
<td>BusinessQuarterEnd</td>
<td>Quarterly dates anchored on last weekday day of each month, for year ending in indicated month</td>
</tr>
</tbody>
</table>
Resampling

• Could be
  - downsampling: higher frequency to lower frequency
  - upsampling: lower frequency to higher frequency
  - neither: e.g. Wednesdays to Fridays
• resample method: e.g. `ts.resample('M').mean()`

<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>freq</td>
<td>String or DateOffset indicating desired resampled frequency (e.g., 'M', '5min', or Second(15))</td>
</tr>
<tr>
<td>axis</td>
<td>Axis to resample on; default axis=0</td>
</tr>
<tr>
<td>fill_method</td>
<td>How to interpolate when upsampling, as in 'ffill' or 'bfill'; by default does no interpolation</td>
</tr>
<tr>
<td>closed</td>
<td>In downsampling, which end of each interval is closed (inclusive), 'right' or 'left'</td>
</tr>
<tr>
<td>label</td>
<td>In downsampling, how to label the aggregated result, with the 'right' or 'left' bin edge (e.g., the 9:30 to 9:35 five-minute interval could be labeled 9:30 or 9:35)</td>
</tr>
<tr>
<td>loffset</td>
<td>Time adjustment to the bin labels, such as '-1s' / Second(-1) to shift the aggregate labels one second earlier</td>
</tr>
<tr>
<td>limit</td>
<td>When forward or backward filling, the maximum number of periods to fill</td>
</tr>
<tr>
<td>kind</td>
<td>Aggregate to periods ('period') or timestamps ('timestamp'); defaults to the type of index the time series has</td>
</tr>
<tr>
<td>convention</td>
<td>When resampling periods, the convention ('start' or 'end') for converting the low-frequency period to high frequency; defaults to 'end'</td>
</tr>
</tbody>
</table>

[W. McKinney, Python for Data Analysis]
Downsampling

- Need to define **bin edges** which are used to group the time series into **intervals** that can be aggregated

- Remember:
  - Which side of the interval is closed
  - How to label the aggregated bin (start or end of interval)

![Diagram showing closed left and right intervals with labels](Image)

**Example Code**

```python
In [219]: ts.resample('5min', closed='right', label='right', loffset=-1s)
```

**Output**

```text
Out [219]:
```

```python
In [220]: ts.resample('5min').ohlc()
```

**Output**

```text
```

**Open-High-Low-Close (OHLC) resampling**

In finance, a popular way to aggregate a time series is to compute four values for each bucket: the first (open), last (close), maximum (high), and minimal (low) values. By using the `ohlc` aggregate function, you will obtain a DataFrame having columns containing these four aggregates, which are efficiently computed in a single sweep of the data:

```python
In [220]: ts.resample('5min').ohlc()
```

**Output**

```text
```

**11.6 Resampling and Frequency Conversion | 351**
Upsampling

• No aggregation necessary

In [222]: frame
Out[222]:

<table>
<thead>
<tr>
<th></th>
<th>Colorado</th>
<th>Texas</th>
<th>New York</th>
<th>Ohio</th>
</tr>
</thead>
<tbody>
<tr>
<td>2000-01-05</td>
<td>-0.896431</td>
<td>0.677263</td>
<td>0.036503</td>
<td>0.087102</td>
</tr>
<tr>
<td>2000-01-12</td>
<td>-0.046662</td>
<td>0.927238</td>
<td>0.482284</td>
<td>-0.867130</td>
</tr>
</tbody>
</table>

In [223]: df_daily = frame.resample('D').asfreq()

In [224]: df_daily
Out[224]:

<table>
<thead>
<tr>
<th></th>
<th>Colorado</th>
<th>Texas</th>
<th>New York</th>
<th>Ohio</th>
</tr>
</thead>
<tbody>
<tr>
<td>2000-01-05</td>
<td>NaN</td>
<td>NaN</td>
<td>NaN</td>
<td>NaN</td>
</tr>
<tr>
<td>2000-01-06</td>
<td>NaN</td>
<td>NaN</td>
<td>NaN</td>
<td>NaN</td>
</tr>
<tr>
<td>2000-01-07</td>
<td>NaN</td>
<td>NaN</td>
<td>NaN</td>
<td>NaN</td>
</tr>
<tr>
<td>2000-01-08</td>
<td>NaN</td>
<td>NaN</td>
<td>NaN</td>
<td>NaN</td>
</tr>
<tr>
<td>2000-01-09</td>
<td>NaN</td>
<td>NaN</td>
<td>NaN</td>
<td>NaN</td>
</tr>
<tr>
<td>2000-01-10</td>
<td>NaN</td>
<td>NaN</td>
<td>NaN</td>
<td>NaN</td>
</tr>
<tr>
<td>2000-01-11</td>
<td>NaN</td>
<td>NaN</td>
<td>NaN</td>
<td>NaN</td>
</tr>
<tr>
<td>2000-01-12</td>
<td>-0.046662</td>
<td>0.927238</td>
<td>0.482284</td>
<td>-0.867130</td>
</tr>
</tbody>
</table>

In [225]: frame.resample('D').ffill()
Out[225]:

<table>
<thead>
<tr>
<th></th>
<th>Colorado</th>
<th>Texas</th>
<th>New York</th>
<th>Ohio</th>
</tr>
</thead>
<tbody>
<tr>
<td>2000-01-05</td>
<td>-0.896431</td>
<td>0.677263</td>
<td>0.036503</td>
<td>0.087102</td>
</tr>
<tr>
<td>2000-01-06</td>
<td>-0.896431</td>
<td>0.677263</td>
<td>0.036503</td>
<td>0.087102</td>
</tr>
<tr>
<td>2000-01-07</td>
<td>-0.896431</td>
<td>0.677263</td>
<td>0.036503</td>
<td>0.087102</td>
</tr>
<tr>
<td>2000-01-08</td>
<td>-0.896431</td>
<td>0.677263</td>
<td>0.036503</td>
<td>0.087102</td>
</tr>
<tr>
<td>2000-01-09</td>
<td>-0.896431</td>
<td>0.677263</td>
<td>0.036503</td>
<td>0.087102</td>
</tr>
<tr>
<td>2000-01-10</td>
<td>-0.896431</td>
<td>0.677263</td>
<td>0.036503</td>
<td>0.087102</td>
</tr>
<tr>
<td>2000-01-11</td>
<td>-0.896431</td>
<td>0.677263</td>
<td>0.036503</td>
<td>0.087102</td>
</tr>
<tr>
<td>2000-01-12</td>
<td>-0.046662</td>
<td>0.927238</td>
<td>0.482284</td>
<td>-0.867130</td>
</tr>
</tbody>
</table>
Rolling Window Calculations

\[ \begin{align*}
12 & \quad 8 & \quad 7 & \quad 4 & \quad 9 & \quad 13 & \quad 4 & \quad 11 & \quad 3 & \quad 8
\end{align*} \]
Rolling Window Calculations

\[
\begin{array}{cccccccc}
12 & 8 & 7 & 4 & 9 & 13 & 4 & 11 & 3 & 8 \\
\end{array}
\]

7.8
Rolling Window Calculations

\[ \begin{array}{cccccc}
12 & 8 & 7 & 4 & 9 & 13 & 4 & 11 & 3 & 8 \\
\end{array} \]

7.8
Rolling Window Calculations

12 8 7 4 9 13 4 11 3 8

7.8 7.0
Rolling Window Calculations

\[
\begin{array}{ccccccc}
12 & 8 & 7 & 4 & 9 & 13 & 4 & 11 & 3 & 8 \\
\end{array}
\]

7.8 7.0
Rolling Window Calculations

12 8 7 4 9 13 4 11 3 8

7.8 7.0 8.3
Window Functions

• Idea: want to aggregate over a window of time, calculate the answer, and then slide that window ahead. Repeat.

• rolling: smooth out data

• Specify the window size in rolling, then an aggregation method

• Result is set to the right edge of window (change with center=True)

• Example:
  - df.rolling('180D').mean()
  - df.rolling('90D').sum()
Shampoo Sales Example
Interpolation

• Fill in the missing values with computed best estimates using various types of algorithms
• Apply after resample
Sales Data by Month
Resampled Sales Data (ffill)
Resampled with Linear Interpolation (Default)
Piecewise Cubic Hermite Interpolating Polynomial
90-Day Rolling Window (Mean)
180-Day Rolling Window (Mean)