DSC 201: Data Analysis & Visualization

Data Aggregation & Time Series

Dr. David Koop
Split-Apply-Combine

- Coined by H. Wickham, 2011
- Similar to Map (split+apply) Reduce (combine) paradigm
- The Pattern:
  1. **Split** the data by some grouping variable
  2. **Apply** some function to each group independently
  3. **Combine** the data into some output dataset
- The apply step is usually one of:
  - Aggregate
  - Transform
  - Filter
Aggregation of time series data, a special use case of `groupby`, is referred to as **resampling** in this book and will receive separate treatment in Chapter 10.

**GroupBy Mechanics**

Hadley Wickham, an author of many popular packages for the R programming language, coined the term **split-apply-combine** for talking about group operations, and I think that's a good description of the process. In the first stage of the process, data contained in a pandas object, whether a Series, DataFrame, or otherwise, is **split** into groups based on one or more **keys** that you provide. The splitting is performed on a particular axis of an object. For example, a DataFrame can be grouped on its rows (`axis=0`) or its columns (`axis=1`). Once this is done, a function is **applied** to each group, producing a new value. Finally, the results of all those function applications are **combined** into a result object. The form of the resulting object will usually depend on what's being done to the data. See Figure 9-1 for a mockup of a simple group aggregation.

**Figure 9-1. Illustration of a group aggregation**

Each grouping key can take many forms, and the keys do not have to be all of the same type:

- A list or array of values that is the same length as the axis being grouped
- A value indicating a column name in a DataFrame

---

[W. McKinney, *Python for Data Analysis*]
Splitting by Variables

<table>
<thead>
<tr>
<th>name</th>
<th>age</th>
<th>sex</th>
</tr>
</thead>
<tbody>
<tr>
<td>John</td>
<td>13</td>
<td>Male</td>
</tr>
<tr>
<td>Mary</td>
<td>15</td>
<td>Female</td>
</tr>
<tr>
<td>Alice</td>
<td>14</td>
<td>Female</td>
</tr>
<tr>
<td>Peter</td>
<td>13</td>
<td>Male</td>
</tr>
<tr>
<td>Roger</td>
<td>14</td>
<td>Male</td>
</tr>
<tr>
<td>Phyllis</td>
<td>13</td>
<td>Female</td>
</tr>
</tbody>
</table>

-.sex-

<table>
<thead>
<tr>
<th>name</th>
<th>age</th>
<th>sex</th>
</tr>
</thead>
<tbody>
<tr>
<td>John</td>
<td>13</td>
<td>Male</td>
</tr>
<tr>
<td>Peter</td>
<td>13</td>
<td>Male</td>
</tr>
<tr>
<td>Roger</td>
<td>14</td>
<td>Male</td>
</tr>
<tr>
<td>Phyllis</td>
<td>13</td>
<td>Female</td>
</tr>
</tbody>
</table>

-.age-

<table>
<thead>
<tr>
<th>name</th>
<th>age</th>
<th>sex</th>
</tr>
</thead>
<tbody>
<tr>
<td>Mary</td>
<td>15</td>
<td>Female</td>
</tr>
<tr>
<td>Alice</td>
<td>14</td>
<td>Female</td>
</tr>
<tr>
<td>Roger</td>
<td>14</td>
<td>Male</td>
</tr>
<tr>
<td>Phyllis</td>
<td>13</td>
<td>Female</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>name</th>
<th>age</th>
<th>sex</th>
</tr>
</thead>
<tbody>
<tr>
<td>Mary</td>
<td>15</td>
<td>Female</td>
</tr>
</tbody>
</table>

[H. Wickham, 2011]
Apply+Combine: Counting

.(sex)

<table>
<thead>
<tr>
<th>sex</th>
<th>value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Male</td>
<td>3</td>
</tr>
<tr>
<td>Female</td>
<td>3</td>
</tr>
</tbody>
</table>

.(age)

<table>
<thead>
<tr>
<th>age</th>
<th>value</th>
</tr>
</thead>
<tbody>
<tr>
<td>13</td>
<td>3</td>
</tr>
<tr>
<td>14</td>
<td>2</td>
</tr>
<tr>
<td>15</td>
<td>1</td>
</tr>
</tbody>
</table>

.(sex, age)

<table>
<thead>
<tr>
<th>sex</th>
<th>age</th>
<th>value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Male</td>
<td>13</td>
<td>2</td>
</tr>
<tr>
<td>Male</td>
<td>14</td>
<td>1</td>
</tr>
<tr>
<td>Female</td>
<td>13</td>
<td>1</td>
</tr>
<tr>
<td>Female</td>
<td>14</td>
<td>1</td>
</tr>
<tr>
<td>Female</td>
<td>15</td>
<td>1</td>
</tr>
</tbody>
</table>

[H. Wickham, 2011]
In Pandas

- **groupby** method creates a GroupBy object
- **groupby** doesn't actually compute anything until there is an apply/aggregate step or we wish to examine the groups
- Choose keys (columns) to group by
- **size()**: size of the groups
- Aggregation Operations:
  - `count()`
  - `mean()`
  - `sum()`
- Can write own function for aggregation and pass it to `agg` function

```python
def peak_to_peak(arr):
    return arr.max() - arr.min()
grouped.agg(peak_to_peak)
```
Assignment 5

- Aggregation, Time Series, and Visualization
- Compare Hurricane Joaquin and Hurricane Maria
Types of GroupBy

• Aggregation: \texttt{agg}
  - \texttt{n:1} n group values become one value
  - Examples: mean, min, median

• Apply: \texttt{apply}
  - \texttt{n:m} n group values become m values
  - Most general (could do aggregation or transform with apply)
  - Example: top 5 in each group
  - Filter

• Transform: \texttt{transform}
  - \texttt{n:n} n group values become n values
  - Cannot mutate the input
### Transform Example

Let's consider a simple example for illustration:

```python
In [76]: df
def
Out[76]:
<table>
<thead>
<tr>
<th>key</th>
<th>value</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>a</td>
</tr>
<tr>
<td>1</td>
<td>b</td>
</tr>
<tr>
<td>2</td>
<td>c</td>
</tr>
<tr>
<td>3</td>
<td>a</td>
</tr>
<tr>
<td>4</td>
<td>b</td>
</tr>
<tr>
<td>5</td>
<td>c</td>
</tr>
<tr>
<td>6</td>
<td>a</td>
</tr>
<tr>
<td>7</td>
<td>b</td>
</tr>
<tr>
<td>8</td>
<td>c</td>
</tr>
<tr>
<td>9</td>
<td>a</td>
</tr>
<tr>
<td>10</td>
<td>b</td>
</tr>
<tr>
<td>11</td>
<td>c</td>
</tr>
</tbody>
</table>
```

In [77]: g = df.groupby('key').value

```python
In [78]: g.mean()
Out[78]:
<table>
<thead>
<tr>
<th>key</th>
<th>value</th>
</tr>
</thead>
<tbody>
<tr>
<td>a</td>
<td>4.5</td>
</tr>
<tr>
<td>b</td>
<td>5.5</td>
</tr>
<tr>
<td>c</td>
<td>6.5</td>
</tr>
</tbody>
</table>
```

In [79]: g.transform(lambda x: x.mean())

```python
Out[79]:
| 0   | 4.5   |
| 1   | 5.5   |
| 2   | 6.5   |
| 3   | 4.5   |
| 4   | 5.5   |
| 5   | 6.5   |
| 6   | 4.5   |
| 7   | 5.5   |
| 8   | 6.5   |
| 9   | 4.5   |
| 10  | 5.5   |
| 11  | 6.5   |
```

[W. McKinney, Python for Data Analysis]
Transform Example

In [76]: df
Out[76]:
   key  value
0    a    0.0
1    b    1.0
2    c    2.0
3    a    3.0
4    b    4.0
5    c    5.0
6    a    6.0
7    b    7.0
8    c    8.0
9    a    9.0
10   b   10.0
11   c   11.0

In [77]: g = df.groupby('key').value

In [78]: g.mean()
Out[78]:
key
a    4.5
b    5.5
c    6.5
Name: value, dtype: float64

In [79]: g.transform(lambda x: x.mean())
Out[79]:
   0    4.5
   1    5.5
   2    6.5
   3    4.5
   4    5.5
   5    6.5
   6    4.5
   7    5.5
   8    6.5
   9    4.5
  10    5.5
  11    6.5
Name: value, dtype: float64

Or g.transform('mean')

[W. McKinney, Python for Data Analysis]
As a more complicated example, we can compute the ranks in descending order for each group:

We can obtain equivalent results in this case either using:

```
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
```
Normalization

```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

[W. McKinney]
Other Operations

• Quantiles: return values at particular splits
  - Median is a 0.5-quantile
  - `df.quantile(0.1)`
  - also works on groups

• Can return data from group-by without having the keys in the index `(as_index=False)` or use `reset_index` after computing

• Grouped weighted average via apply
Pivot Tables

- Data summarization tool in many spreadsheet programs
- Aggregates a table of data by one or more keys with some keys arranged on rows (index), others as columns (columns)
- Pandas supports via `pivot_table` method
- `margins=True` gives partial totals
- Can use different aggregation functions via `aggfunc` kwarg

<table>
<thead>
<tr>
<th>Function name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>values</td>
<td>Column name or names to aggregate. By default aggregates all numeric columns</td>
</tr>
<tr>
<td>rows</td>
<td>Column names or other group keys to group on the rows of the resulting pivot table</td>
</tr>
<tr>
<td>cols</td>
<td>Column names or other group keys to group on the columns of the resulting pivot table</td>
</tr>
<tr>
<td>aggfunc</td>
<td>Aggregation function or list of functions; 'mean' by default. Can be any function valid in a groupby context</td>
</tr>
<tr>
<td>fill_value</td>
<td>Replace missing values in result table</td>
</tr>
<tr>
<td>margins</td>
<td>Add row/column subtotals and grand total, False by default</td>
</tr>
</tbody>
</table>

[W. McKinney, Python for Data Analysis]
Pivot Tables in Pandas

• **tips**

<table>
<thead>
<tr>
<th></th>
<th>total_bill</th>
<th>tip</th>
<th>sex</th>
<th>smoker</th>
<th>day</th>
<th>time</th>
<th>size</th>
<th>tip_pct</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>16.99</td>
<td>1.01</td>
<td>Female</td>
<td>No</td>
<td>Sun</td>
<td>Dinner</td>
<td>2</td>
<td>0.059447</td>
</tr>
<tr>
<td>1</td>
<td>10.34</td>
<td>1.66</td>
<td>Male</td>
<td>No</td>
<td>Sun</td>
<td>Dinner</td>
<td>3</td>
<td>0.160542</td>
</tr>
<tr>
<td>2</td>
<td>21.01</td>
<td>3.50</td>
<td>Male</td>
<td>No</td>
<td>Sun</td>
<td>Dinner</td>
<td>3</td>
<td>0.166587</td>
</tr>
<tr>
<td>3</td>
<td>23.68</td>
<td>3.31</td>
<td>Male</td>
<td>No</td>
<td>Sun</td>
<td>Dinner</td>
<td>2</td>
<td>0.139780</td>
</tr>
<tr>
<td>4</td>
<td>24.59</td>
<td>3.61</td>
<td>Female</td>
<td>No</td>
<td>Sun</td>
<td>Dinner</td>
<td>4</td>
<td>0.146808</td>
</tr>
<tr>
<td>5</td>
<td>25.29</td>
<td>4.71</td>
<td>Male</td>
<td>No</td>
<td>Sun</td>
<td>Dinner</td>
<td>4</td>
<td>0.186240</td>
</tr>
<tr>
<td>6</td>
<td>8.77</td>
<td>2.00</td>
<td>Male</td>
<td>No</td>
<td>Sun</td>
<td>Dinner</td>
<td>2</td>
<td>0.228050</td>
</tr>
</tbody>
</table>

• **tips.pivot_table(index=['sex', 'smoker'])**

<table>
<thead>
<tr>
<th></th>
<th>size</th>
<th>tip</th>
<th>tip_pct</th>
<th>total_bill</th>
</tr>
</thead>
<tbody>
<tr>
<td>sex</td>
<td>smoker</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Female</td>
<td>No</td>
<td>2.592593</td>
<td>2.773519</td>
<td>0.156921</td>
</tr>
<tr>
<td></td>
<td>Yes</td>
<td>2.242424</td>
<td>2.931515</td>
<td>0.182150</td>
</tr>
<tr>
<td>Male</td>
<td>No</td>
<td>2.711340</td>
<td>3.113402</td>
<td>0.160669</td>
</tr>
<tr>
<td></td>
<td>Yes</td>
<td>2.500000</td>
<td>3.051167</td>
<td>0.152771</td>
</tr>
</tbody>
</table>
Pivot Tables with Margins and Aggfunc

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

<table>
<thead>
<tr>
<th>sex</th>
<th>smoker</th>
<th>day</th>
<th>size</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>No</td>
<td>Fri</td>
<td>2.0</td>
</tr>
<tr>
<td></td>
<td>Yes</td>
<td>2.0</td>
<td>7.0</td>
</tr>
<tr>
<td></td>
<td>All</td>
<td>2.0</td>
<td>9.0</td>
</tr>
<tr>
<td></td>
<td>No</td>
<td>Sat</td>
<td>13.0</td>
</tr>
<tr>
<td></td>
<td>Yes</td>
<td>13.0</td>
<td>15.0</td>
</tr>
<tr>
<td>Female</td>
<td>All</td>
<td>28.0</td>
<td></td>
</tr>
<tr>
<td></td>
<td>No</td>
<td>Sun</td>
<td>14.0</td>
</tr>
<tr>
<td></td>
<td>Yes</td>
<td>14.0</td>
<td>4.0</td>
</tr>
<tr>
<td></td>
<td>All</td>
<td>14.0</td>
<td>18.0</td>
</tr>
<tr>
<td></td>
<td>No</td>
<td>Thur</td>
<td>25.0</td>
</tr>
<tr>
<td></td>
<td>Yes</td>
<td>25.0</td>
<td>7.0</td>
</tr>
<tr>
<td></td>
<td>All</td>
<td>25.0</td>
<td>32.0</td>
</tr>
<tr>
<td></td>
<td>No</td>
<td>Fri</td>
<td>2.0</td>
</tr>
<tr>
<td>Male</td>
<td>Yes</td>
<td>2.0</td>
<td>8.0</td>
</tr>
<tr>
<td></td>
<td>All</td>
<td>2.0</td>
<td>10.0</td>
</tr>
<tr>
<td></td>
<td>No</td>
<td>Sat</td>
<td>32.0</td>
</tr>
<tr>
<td></td>
<td>Yes</td>
<td>32.0</td>
<td>27.0</td>
</tr>
<tr>
<td></td>
<td>All</td>
<td>32.0</td>
<td>59.0</td>
</tr>
<tr>
<td></td>
<td>No</td>
<td>Sun</td>
<td>43.0</td>
</tr>
<tr>
<td></td>
<td>Yes</td>
<td>43.0</td>
<td>15.0</td>
</tr>
<tr>
<td></td>
<td>All</td>
<td>43.0</td>
<td>58.0</td>
</tr>
<tr>
<td></td>
<td>No</td>
<td>Thur</td>
<td>20.0</td>
</tr>
<tr>
<td></td>
<td>Yes</td>
<td>20.0</td>
<td>10.0</td>
</tr>
<tr>
<td></td>
<td>All</td>
<td>20.0</td>
<td>30.0</td>
</tr>
<tr>
<td></td>
<td>No</td>
<td>All</td>
<td>151.0</td>
</tr>
<tr>
<td></td>
<td>Yes</td>
<td>151.0</td>
<td>93.0</td>
</tr>
<tr>
<td></td>
<td>All</td>
<td>151.0</td>
<td>244.0</td>
</tr>
</tbody>
</table>

D. Koop, DSC 201, Fall 2018

14
Crosstabs

• **crosstab** is a special case for group frequencies (aggfunc='count')

  In [293]: pd.crosstab(data.Gender, data.Handedness, margins=True)
  Out[293]:
  Handedness       Left-handed  Right-handed  All
  Gender          
  Female          1             4          5
  Male            2             3          5
  All             3             7          10

• Tipping example
• Also see the Federal Election Database example in the book
Crosstab

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

<table>
<thead>
<tr>
<th>smoker</th>
<th>No</th>
<th>Yes</th>
<th>All</th>
</tr>
</thead>
<tbody>
<tr>
<td>time</td>
<td>day</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Dinner</td>
<td>Fri</td>
<td>3</td>
<td>9</td>
</tr>
<tr>
<td></td>
<td>Sat</td>
<td>45</td>
<td>42</td>
</tr>
<tr>
<td></td>
<td>Sun</td>
<td>57</td>
<td>19</td>
</tr>
<tr>
<td></td>
<td>Thur</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>Lunch</td>
<td>Fri</td>
<td>1</td>
<td>6</td>
</tr>
<tr>
<td></td>
<td>Thur</td>
<td>44</td>
<td>17</td>
</tr>
<tr>
<td>All</td>
<td>151</td>
<td>93</td>
<td>244</td>
</tr>
</tbody>
</table>

• Or... tips.pivot_table('total_bill',index=['time', 'day'], columns=['smoker'], aggfunc='count', margins=True, fill_value=0)
Example: Tipping Data

- http://www.cis.umassd.edu/~dkoop/dsc201-2018fa/notebooks/tipping.ipynb
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)
Dates and Times

• What is time to a computer?
  - Can be stored as seconds since Unix Epoch (January 1st, 1970)
• Often useful to break down into minutes, hours, days, months, years…
• Lots of different ways to write time:
  - How could you write "November 29, 2016"?
  - European vs. American ordering…
• What about time zones?
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.
### Datetime format specification

- **Look it up:**
  - [http://strftime.org](http://strftime.org)

- Generally, can create whatever format you need using these format strings

<table>
<thead>
<tr>
<th>Code</th>
<th>Meaning</th>
<th>Example</th>
</tr>
</thead>
<tbody>
<tr>
<td>%a</td>
<td>Weekday as locale’s abbreviated name.</td>
<td>Mon</td>
</tr>
<tr>
<td>%A</td>
<td>Weekday as locale’s full name.</td>
<td>Monday</td>
</tr>
<tr>
<td>%w</td>
<td>Weekday as a decimal number, where 0 is Sunday and 6 is Saturday.</td>
<td>1</td>
</tr>
<tr>
<td>%d</td>
<td>Day of the month as a zero-padded decimal number.</td>
<td>30</td>
</tr>
<tr>
<td>%m</td>
<td>Month as a zero-padded decimal number.</td>
<td>09</td>
</tr>
<tr>
<td>%y</td>
<td>Year without century as a zero-padded decimal number.</td>
<td>13</td>
</tr>
<tr>
<td>%Y</td>
<td>Year with century as a decimal number.</td>
<td>2013</td>
</tr>
<tr>
<td>%H</td>
<td>Hour (24-hour clock) as a zero-padded decimal number.</td>
<td>07</td>
</tr>
<tr>
<td>%I</td>
<td>Hour (12-hour clock) as a zero-padded decimal number.</td>
<td>07</td>
</tr>
<tr>
<td>%p</td>
<td>Locale’s equivalent of either AM or PM.</td>
<td>AM</td>
</tr>
<tr>
<td>%M</td>
<td>Minute as a zero-padded decimal number.</td>
<td>06</td>
</tr>
<tr>
<td>%m</td>
<td>Minute as a decimal number. (Platform specific)</td>
<td>6</td>
</tr>
<tr>
<td>%S</td>
<td>Second as a zero-padded decimal number.</td>
<td>05</td>
</tr>
<tr>
<td>%s</td>
<td>Second as a decimal number. (Platform specific)</td>
<td>5</td>
</tr>
</tbody>
</table>
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 (like **NaN** but for timestamps)
- Stores in a **numpy.datetime64 format**
- **pd.Timestamp**: a wrapper for the **datetime64 objects**
- Can use time as an index
- Accessing a particular time or checking equivalence allows any string that can be interpreted as a date:
  - `ts['1/10/2011']` or `ts['20110110']`
More Pandas Support

• Slicing works as expected
• Can do operations (add, subtract) on data indexed by datetime and the indexes will match up
• As with strings, to treat a column as datetime, you can use the .dt accessor
Generating Date Ranges

- `index = pd.date_range('4/1/2012', '6/1/2012')`
- Can generate based on a number of periods as well
  - `index = pd.date_range('4/1/2012', periods=20)`
- Frequency (`freq`) controls how the range is divided
  - Codes for specifying this (e.g. 4h, D, M)
    - In [90]: `pd.date_range('1/1/2000', '1/3/2000 23:59', freq='4h')`
    - `Out[90]`:
      - `<class 'pandas.tseries.index.DatetimeIndex'>`
      - `[2000-01-01 00:00:00, ..., 2000-01-03 20:00:00]`
      - Length: 18, Freq: 4H, Timezone: None
    - Can also mix them: '2h30m'
Some frequencies describe points in time that are not evenly spaced. For example, ‘M’ (calendar month end) and ‘BM’ (last business/weekday of month) depend on the number of days in a month and, in the latter case, whether the month ends on a weekend or not. For lack of a better term, I call these anchored offsets.

See Table 10-4 for a listing of frequency codes and date offset classes available in pandas. Users can define their own custom frequency classes to provide date logic not available in pandas, though the full details of that are outside the scope of this book.

### Table 10-4. Base Time Series Frequencies

<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 (1/1000th of 1 second)</td>
</tr>
<tr>
<td>U</td>
<td>Micro</td>
<td>Microsecond (1/1000000th 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 day 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. For example, WOM-3FRI for the 3rd Friday of each month.[W. McKinney, Python for Data Analysis]</td>
</tr>
</tbody>
</table>
DatetimeIndex

• Can use time as an index

• 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 Data

• Leading or Lagging Data

In [95]: ts = Series(np.random.randn(4),
                  index=pd.date_range('1/1/2000', periods=4, freq='M'))

In [96]: ts
Out[96]:
2000-01-31   -0.066748
2000-02-29    0.838639
2000-03-31   -0.117388
2000-04-30   -0.517795
Freq: M, dtype: float64

In [97]: ts.shift(2)
Out[97]:
2000-03-31   -0.066748
2000-04-30    0.838639
2000-05-31   -0.117388
2000-06-30   -0.517795
Freq: M, dtype: float64

In [98]: ts.shift(-2)
Out[98]:
2000-01-31 01:30:00   -0.066748
2000-02-29 01:30:00    0.838639
2000-03-31 01:30:00   -0.117388
2000-04-30 01:30:00   -0.517795
Freq: M, dtype: float64

• Shifting by time:

In [99]: ts.shift(2, freq='M')
Out[99]:
2000-03-31   -0.066748
2000-04-30    0.838639
2000-05-31   -0.117388
2000-06-30   -0.517795
Freq: M, dtype: float64
Shifting Time Series

• Data:
  ```
  [('2017-11-30', 48), ('2017-12-02', 45),
   ('2017-12-03', 44), ('2017-12-04', 48)]
  ```

• Compute day-to-day difference in high temperature:
  ```
  - s - s.shift(1)  (same as s.diff())

  - 2017-11-30   NaN
  2017-12-02    -3.0
  2017-12-03    -1.0
  2017-12-04    4.0

  - s - s.shift(1, 'd')

  - 2017-11-30   NaN
  2017-12-01    NaN
  2017-12-02    NaN
  2017-12-03    -1.0
  2017-12-04    4.0
  2017-12-05    NaN
  ```
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 take strings:
  ```python
datetime.now().date() + pd.Timedelta("4 days")
  ```
- Also, Roll dates using anchored offsets

Shifting dates with offsets

The pandas date offsets can also be used with datetime or Timestamp objects:

```python
In [103]: from pandas.tseries.offsets import Day, MonthEnd
In [104]: now = datetime(2011, 11, 17)
In [105]: now + 3 * Day()
Out[105]: Timestamp('2011-11-20 00:00:00')
```

If you add an anchored offset like MonthEnd, the first increment will roll forward a date to the next date according to the frequency rule:

```python
In [106]: now + MonthEnd()
Out[106]: Timestamp('2011-11-30 00:00:00')
In [107]: now + MonthEnd(2)
Out[107]: Timestamp('2011-12-31 00:00:00')
```

Anchored offsets can explicitly “roll” dates forward or backward using their rollforward and rollback methods, respectively:

```python
In [108]: offset = MonthEnd()
In [109]: offset.rollforward(now)
Out[109]: Timestamp('2011-11-30 00:00:00')
```

298 | Chapter 10: Time Series