Advanced Data Management (CSCI 680/490)

Time Series Data

Dr. David Koop
Spatial Data

Measure vegetation density

Track hurricanes

Track phytoplankton populations

Measure snow melt
Interactive Exploration of Spatial Data

```
SELECT lat, lng, (b4-b6)/(b4+b6) as ndsi
FROM modis_data
WHERE ndsi > 0.7
```
Interactive Exploration of Spatial Data

SELECT lat, lng, (b4-b6)/(b4+b6) as ndsi
FROM modis_data
WHERE ndsi > 0.7
Two Inputs to Exploratory Browsing

- User submits query
- Prepare data in DBMS (Pre-comp. Structures)
- Create visualization
- User pan/zoom
- Fetch results from DBMS
- Update visualization

Cold start time: interaction latency < 500ms

[L. Battle, 2017]
### Systems for Interactive Exploration

<table>
<thead>
<tr>
<th>Output format</th>
<th>Time</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Pre-computed structures (Offline)</td>
</tr>
<tr>
<td>Sampling</td>
<td>Predictive (Before interaction)</td>
</tr>
<tr>
<td>Aggregation</td>
<td>Progressive/Incremental (After interaction)</td>
</tr>
</tbody>
</table>

- **Sampling**
  - Nanocubes (Infovis 2013)
  - imMens (Eurovis 2013)
- **Aggregation**
  - ATLAS (VAST 2008)
  - XmdvTool (DASFAA 2003)
- **Output format**
  - ForeCache

**Time**

- SampleAction (CHI 2012)
- Vizdom (VLDB 2015)
- DICE (ICDE 2014)
- A-WARE (HILDA 2016)

[L. Battle, 2017]
Nanocubes

Linked view of tweets in San Diego, US [Lins et. al, 2013]
From Tables and Spreadsheets to Data Cubes

- A **data warehouse** is based on a multidimensional data model which views data in the form of a data cube.
- A **data cube**, such as sales, allows data to be modeled and viewed in multiple dimensions:
  - **Dimension tables**, such as item (item_name, brand, type), or time(day, week, month, quarter, year)
  - **Fact table** contains **measures** (such as dollars_sold) and keys to each of the related dimension tables.
- In data warehousing literature, an n-D base cube is called a **base cuboid**. The top most 0-D cuboid, which holds the highest-level of summarization, is called the **apex cuboid**. The lattice of cuboids forms a **data cube**.

[Han et al., 2011]
Data Cube: A Lattice of Cuboids

0-D (apex) cuboid

1-D cuboids

2-D cuboids

3-D cuboids

4-D (base) cuboid

[Han et al., 2011]
Data Cube Measures: Three Categories

- **Distributive**: if the result derived by applying the function to n aggregate values is the same as that derived by applying the function on all the data without partitioning
  - E.g., `count()`, `sum()`, `min()`, `max()`

- **Algebraic**: if it can be computed by an algebraic function with M arguments (where M is a bounded integer), each of which is obtained by applying a distributive aggregate function
  - E.g., `avg()`, `min_N()`, `standard_deviation()`

- **Holistic**: if there is no constant bound on the storage size needed to describe a subaggregate.
  - E.g., `median()`, `mode()`, `rank()`
Multidimensional Data

- Sales volume as a function of product, month, and region

Dimensions: Product, Location, Time
Hierarchical summarization paths
A Sample Data Cube

Total annual sales of TVs in U.S.A.

Product: TV, VCR, PC

Date: 1Qtr, 2Qtr, 3Qtr, 4Qtr, sum

Country: U.S.A, Canada, Mexico, sum

sum, sum, sum

[Han et al., 2011]
OLAP Operations

[Han et al., 2011]
Efficient Processing of OLAP Queries

• Determine which operations should be performed on the available cuboids
  - Transform drill, roll, etc. into corresponding SQL and/or OLAP operations, e.g., dice = selection + projection

• Determine which materialized cuboid(s) for OLAP operation:
  - Query: \{brand, province_or_state\} with “year = 2004”
  - 4 materialized cuboids available:
    1. \{year, item_name, city\}
    2. \{year, brand, country\}
    3. \{year, brand, province_or_state\}
    4. \{item_name, province_or_state\} where year = 2004

  - Which should be selected to process the query?

[Han et al., 2011]
Data Cube Aggregations

Relation A

<table>
<thead>
<tr>
<th>Country</th>
<th>Device</th>
<th>Language</th>
</tr>
</thead>
<tbody>
<tr>
<td>US</td>
<td>Android</td>
<td>en</td>
</tr>
<tr>
<td>US</td>
<td>iPhone</td>
<td>ru</td>
</tr>
<tr>
<td>South Africa</td>
<td>iPhone</td>
<td>en</td>
</tr>
<tr>
<td>India</td>
<td>Android</td>
<td>en</td>
</tr>
<tr>
<td>Australia</td>
<td>iPhone</td>
<td>en</td>
</tr>
</tbody>
</table>

Aggregation B

<table>
<thead>
<tr>
<th>Country</th>
<th>Device</th>
<th>Language</th>
<th>Count</th>
</tr>
</thead>
<tbody>
<tr>
<td>All</td>
<td>All</td>
<td>All</td>
<td>5</td>
</tr>
</tbody>
</table>

Group By on Device, Language C

<table>
<thead>
<tr>
<th>Country</th>
<th>Device</th>
<th>Language</th>
<th>Count</th>
</tr>
</thead>
<tbody>
<tr>
<td>All</td>
<td>Android</td>
<td>en</td>
<td>2</td>
</tr>
<tr>
<td>All</td>
<td>iPhone</td>
<td>en</td>
<td>2</td>
</tr>
</tbody>
</table>

Cube on Device, Language D

<table>
<thead>
<tr>
<th>Country</th>
<th>Device</th>
<th>Language</th>
<th>Count</th>
</tr>
</thead>
<tbody>
<tr>
<td>All</td>
<td>All</td>
<td>All</td>
<td>5</td>
</tr>
<tr>
<td>All</td>
<td>Android</td>
<td>All</td>
<td>2</td>
</tr>
<tr>
<td>All</td>
<td>iPhone</td>
<td>All</td>
<td>3</td>
</tr>
<tr>
<td>All</td>
<td>All</td>
<td>en</td>
<td>4</td>
</tr>
<tr>
<td>All</td>
<td>All</td>
<td>ru</td>
<td>1</td>
</tr>
<tr>
<td>All</td>
<td>iPhone</td>
<td>ru</td>
<td>1</td>
</tr>
<tr>
<td>All</td>
<td>Android</td>
<td>en</td>
<td>2</td>
</tr>
<tr>
<td>All</td>
<td>iPhone</td>
<td>en</td>
<td>2</td>
</tr>
</tbody>
</table>

Equivalent to Group By on all possible subsets of \{Device, Language\}

[Lins et. al, 2013]
Building a Nanocube

Five Tweets: Location and Device

1. \( \ell_{\text{device}}( \bigcirc ) = \text{Android} \)
2. \( \ell_{\text{device}}( \bullet ) = \text{iPhone} \)
3. \( \ell_{\text{spatial1}} \)
4. \( \ell_{\text{spatial2}} \)
5. "Five Tweets: Location and Device"

Indexing Schema

\[ S = [ \ell_{\text{spatial1}}, \ell_{\text{spatial2}}, \ell_{\text{device}} ] \]
Assignment 5

- Chicago Bike Sharing Data
  - Spatial Analysis
  - Temporal Analysis
  - Graph Database (neo4j)
Teaching Evaluations

- This Wednesday (April 20) in class
TopKube: A Rank-Aware Data Cube for Real-Time Exploration of Spatiotemporal Data

F. Miranda, L. Lins, J. T. Klosowski, and C. T. Silva
TopKube: What about Top-k and Rankings?

- Aggregates are interesting
- Also, often interested in \textit{top-k} answers given particular criteria
- …or \textit{rankings}
- Search over time and space but find specific examples
- TopKube is a rank-aware data structure that computes top-k queries with low latency so interactive exploration is possible
Example: Basketball

- Shots by time, number of points scored, and location on the court

<table>
<thead>
<tr>
<th>team</th>
<th>player</th>
<th>time</th>
<th>pts</th>
<th>x</th>
<th>y</th>
</tr>
</thead>
<tbody>
<tr>
<td>CLE</td>
<td>L. James</td>
<td>5</td>
<td>0</td>
<td>13</td>
<td>28</td>
</tr>
<tr>
<td>BOS</td>
<td>R. Rondo</td>
<td>5</td>
<td>2</td>
<td>38</td>
<td>26</td>
</tr>
<tr>
<td>CLE</td>
<td>L. James</td>
<td>7</td>
<td>3</td>
<td>42</td>
<td>35</td>
</tr>
</tbody>
</table>

- Query: Ranked list of the 50 players who took the most shots
  - `SELECT player, count(*) AS shots FROM table GROUP BY player ORDER BY shots DESC LIMIT 50`

- Query: Rank the top 50 players by points made:
  - `SELECT player, sum(pts) AS points FROM table GROUP BY player ORDER BY points DESC LIMIT 50`
Ranking by Shot Location

![Heatmap of shot locations for NBA players](image)

<table>
<thead>
<tr>
<th>Rank</th>
<th>Name</th>
<th>Shots</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Jamal Crawford</td>
<td>113</td>
</tr>
<tr>
<td>2</td>
<td>Arron Afflalo</td>
<td>105</td>
</tr>
<tr>
<td>3</td>
<td>Rashard Lewis</td>
<td>98</td>
</tr>
<tr>
<td>4</td>
<td>Martell Webster</td>
<td>96</td>
</tr>
<tr>
<td>5</td>
<td>Joe Johnson</td>
<td>86</td>
</tr>
<tr>
<td>6</td>
<td>Rasual Butler</td>
<td>85</td>
</tr>
<tr>
<td>7</td>
<td>Jason Terry</td>
<td>81</td>
</tr>
<tr>
<td>8</td>
<td>Anthony Parker</td>
<td>80</td>
</tr>
<tr>
<td>9</td>
<td>Danilo Gallinari</td>
<td>78</td>
</tr>
<tr>
<td>10</td>
<td>George Hill</td>
<td>75</td>
</tr>
<tr>
<td>11</td>
<td>Ray Allen</td>
<td>69</td>
</tr>
<tr>
<td>12</td>
<td>Steve Blake</td>
<td>68</td>
</tr>
<tr>
<td>13</td>
<td>Mickael Pietrus</td>
<td>68</td>
</tr>
<tr>
<td>14</td>
<td>Mo Williams</td>
<td>63</td>
</tr>
<tr>
<td>15</td>
<td>Keith Bogans</td>
<td>63</td>
</tr>
<tr>
<td>16</td>
<td>Anthony Morrow</td>
<td>62</td>
</tr>
<tr>
<td>17</td>
<td>Mike Bibby</td>
<td>62</td>
</tr>
<tr>
<td>18</td>
<td>Al Harrington</td>
<td>62</td>
</tr>
<tr>
<td>19</td>
<td>Shane Battier</td>
<td>61</td>
</tr>
<tr>
<td>20</td>
<td>Carlos Delfino</td>
<td>61</td>
</tr>
</tbody>
</table>

[D. Koop, CSCI 680/490, Spring 2022]  
{[F. Miranda et al., 2017]}
TopKube vs. Nanocubes

- Product bin: the combination of selections from dimensions
- Nanocubes maps each product bin \(((01,10), \text{iPhone})\) to a time series
  \[
  \beta \mapsto ((t_1, v_1), (t_2, v_1 + v_2), \ldots, (t_m, v_1 + \ldots + v_m))
  \]
- TopKube maps each product bin to rank-aware multi-set
  \[
  \beta \mapsto \left\{ \text{lst} = ((q_1, v_1, \sigma_1), \ldots, (q_j, v_j, \sigma_j)), \text{sum} = \sum_{i=1}^{j} v_i \right\}
  \]
- \(q_i\) is the ith smallest key that appears in product bin
- \(v_i\) is the value of the measure for key \(q_i\) in the product bin
- \(\sigma_i\) is the index of the key with its largest value
Example: One Spatial Dim. and A,B,C events

\[
\begin{array}{cccc}
q & v & o & r \\
B & 1 & 2 & \\
C & 2 & 1 & sum 3
\end{array}
\]

\[
\begin{array}{cccc}
q & v & o & r \\
A & 2 & 1 & \\
C & 1 & 2 & sum 3
\end{array}
\]

\[
\begin{array}{cccc}
q & v & o & r \\
A & 1 & 2 & \\
C & 3 & 1 & sum 4
\end{array}
\]

\[
\begin{array}{cccc}
q & v & o & r \\
A & 3 & 2 & \\
C & 4 & 1 & sum 7
\end{array}
\]

\[
\begin{array}{cccc}
q & v & o & r \\
A & 2 & 1 & \\
B & 1 & 2 & sum 3
\end{array}
\]

\[
\begin{array}{cccc}
q & v & o & r \\
A & 5 & 3 & \\
B & 2 & 1 & sum 13
\end{array}
\]

[Farid Miranda et al., 2017]
Problem: Lots of Bins!

For the temporal dimension, the particular constraint in the time dimension. Note that by taking differ-
ent constraints, we can consider various constraints on the time dimension, the particular
constraint in the time dimension. Note that by taking differ-
ent constraints, we can consider various constraints on the time dimension. Note that by taking differ-
ent constraints, we can consider various constraints on the time dimension. Note that by taking differ-
ent constraints, we can consider various constraints on the time dimension. Note that by taking differ-
ent constraints, we can consider various constraints on the time dimension. Note that by taking differ-
ent constraints, we can consider various constraints on the time dimension.

Conceptually, we can think of Nanocubes as an encoding to a
summed area tables

$$B_{\text{space}}$$

$$B_{\text{time}}$$

$$[3, 6]$$

[F. Miranda et al., 2017]
Three Algorithms to Merge Bins

• Threshold: don't do a full scan, use extra information about ranking
• Sweep: Use a priority queue where the product bin with the current smallest key is on the top
• Hybrid:
  - Threshold has best theoretical guarantee but some sparse cases can be faster
  - Use Sweep on small input lists, Threshold on denser problem
Top-edited Wikipages in Nevada and Mississippi

Wikipedia:
we can see that there was an unusual spike of activity during which we were able to trace their location. The final dataset, with Anonymous edits containing the IP information of the user, (1.57 million unique ones). Figure 8 shows how exploration visualization of the dataset using T
extracted the latitude, longitude, and hashtags from the blog. Available geotagged microblog entries. From each post, we during his bike trip.

Flickr: the high activity spike is mostly due to photos tagged with orange one covering the high activity days. We can see that even though Nevada is not considered a state with a high percentage of religious people, religious articles are among the highest ranked. On the other hand, Mississippi, a state with a high percentage of religious people, contains edit history for every article since its creation in 2005.

Geotagged microblog entries. From each post, we during his bike trip.

TOP-EDITED WIKIPAGES IN NEVADA AND MISSISSIPPI

<table>
<thead>
<tr>
<th>Topic</th>
<th>Nevada</th>
<th>Mississippi</th>
<th>Louisiana</th>
<th>New Orleans Saints</th>
</tr>
</thead>
<tbody>
<tr>
<td>Baton Rouge, Louisiana</td>
<td>303</td>
<td>230</td>
<td>216</td>
<td></td>
</tr>
<tr>
<td>University of Mississippi...</td>
<td></td>
<td>216</td>
<td>208</td>
<td></td>
</tr>
<tr>
<td>Mississippi</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Jackson, Mississippi</td>
<td>208</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Louisiana State University...</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Mississippi State University...</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>WLTV</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Ole Miss Rebels football...</td>
<td>155</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>List of Star Wars books...</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Louisiana</td>
<td>122</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>New Orleans Saints</td>
<td>107</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Reno, Nevada
Early Christianity
Comparison of the AK-47 and M1
Las Vegas Academy
Timeline of Christianity
Las Vegas
Council of Jerusalem
Paul the Apostle
University of Nevada, Las Vegas
Nevada
Antinomianism

(F. Miranda et al., 2017)
Geolocated Flickr tags in Africa

![Geolocation Map of Africa with highlighted geotagged entries](image)

- **Map** showing geotagged entries in Africa with latitude and longitude data.
- **Table** showing hashtag frequencies:
  - `africa`: 652
  - `namibia`: 275
  - `afrique`: 258
  - `senegal`: 253
  - `nigeria`: 198
  - `west_africa`: 156
  - `square`: 125
  - `iphoneography`: 124
  - `instagram_app`: 123
  - `dakar`: 122

- **Graph** showing timeseries data:
  - X-axis: Dates from Jan 1, 2012 to Jan 26, 2014
  - Y-axis: Count of geotagged entries
  - Data points indicate spikes in activity during specific dates.

- **Figure 8** shows how exploration visualization of the dataset using T
- **Figure 7** presents a comparison of the top-edited articles in Nevada and Mississippi.
- **Table** showing top 20 articles in different regions:
  - Nevada: #paris, #charliehebdo
  - Mississippi: #religion

- **Microblogging** and **Flickr** datasets are used for exploration.
- **Wikipedia** and **GitHub** data sources.

- **Performance** analysis of algorithms with **T**.
- **Use Cases** include analysis of unusual spikes and top articles.

- **Techniques** include geolocation, temporal analysis, and hashtag extraction.

- **Results** indicate insights into unusual events and top topics.

---

[Image: [F. Miranda et al., 2017]]
Top Hashtags in Paris related to Charlie Hebdo

1. Select Paris Area


3. Select this Spike and Observe Top-10 Hashtags

4. Select Charlie Hebdo’s Top Hashtags and Observe its Temporal Volume Pattern

[F. Miranda et al., 2017]
GitHub Top commits near urban centers

![GitHub Top commits near urban centers](image-url)

[F. Miranda et al., 2017]
Evaluation

![Graph showing cumulative probability against log10(milliseconds) for different methods]

- PostGIS
- Threshold Algorithm
- Sweep
- Hybrid 0.75
- Hybrid 0.50
- Hybrid 0.25

[F. Miranda et al., 2017]
Aggregation
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
## Split-Apply-Combine

<table>
<thead>
<tr>
<th>key</th>
<th>data</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>0</td>
</tr>
<tr>
<td>B</td>
<td>5</td>
</tr>
<tr>
<td>C</td>
<td>10</td>
</tr>
<tr>
<td>A</td>
<td>5</td>
</tr>
<tr>
<td>B</td>
<td>10</td>
</tr>
<tr>
<td>C</td>
<td>15</td>
</tr>
<tr>
<td>A</td>
<td>10</td>
</tr>
<tr>
<td>B</td>
<td>5</td>
</tr>
<tr>
<td>C</td>
<td>15</td>
</tr>
<tr>
<td>C</td>
<td>20</td>
</tr>
</tbody>
</table>

### Split

<table>
<thead>
<tr>
<th>key</th>
<th>data</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>0</td>
</tr>
<tr>
<td>A</td>
<td>5</td>
</tr>
<tr>
<td>A</td>
<td>10</td>
</tr>
<tr>
<td>B</td>
<td>5</td>
</tr>
<tr>
<td>B</td>
<td>10</td>
</tr>
<tr>
<td>B</td>
<td>15</td>
</tr>
<tr>
<td>C</td>
<td>10</td>
</tr>
<tr>
<td>C</td>
<td>15</td>
</tr>
<tr>
<td>C</td>
<td>20</td>
</tr>
</tbody>
</table>

### Apply

- A: sum of 0, 5, 10 = 15
- B: sum of 5, 10, 15 = 30
- C: sum of 10, 15, 20 = 45

### Combine

<table>
<thead>
<tr>
<th>key</th>
<th>data</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>15</td>
</tr>
<tr>
<td>B</td>
<td>30</td>
</tr>
<tr>
<td>C</td>
<td>45</td>
</tr>
</tbody>
</table>

[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>
</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>
<tr>
<td>Alice</td>
<td>14</td>
<td>Female</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>John</td>
<td>13</td>
<td>Male</td>
</tr>
<tr>
<td>Peter</td>
<td>13</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>
<tr>
<td>Alice</td>
<td>14</td>
<td>Female</td>
</tr>
<tr>
<td>Roger</td>
<td>14</td>
<td>Male</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

<table>
<thead>
<tr>
<th>-sex</th>
<th>(age)</th>
<th>(sex, age)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Male</td>
<td>13</td>
<td>Male 13</td>
</tr>
<tr>
<td></td>
<td>14</td>
<td>Male 14</td>
</tr>
<tr>
<td></td>
<td>15</td>
<td>Female 13</td>
</tr>
<tr>
<td>Female</td>
<td>3</td>
<td>Female 14</td>
</tr>
<tr>
<td></td>
<td>2</td>
<td>Female 15</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()` is the count of each group
Aggregation

- Operations:
  - `count()`
  - `mean()`
  - `sum()`

- May also wish to aggregate only certain subsets
  - Use square brackets with column names

- Can also write your own functions for aggregation and pass them to `agg` function
  - `def peak_to_peak(arr):
    return arr.max() - arr.min()
grouped.agg(peak_to_peak)`
Optimized groupby methods

<table>
<thead>
<tr>
<th>Function name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>count</td>
<td>Number of non-NA values in the group</td>
</tr>
<tr>
<td>sum</td>
<td>Sum of non-NA values</td>
</tr>
<tr>
<td>mean</td>
<td>Mean of non-NA values</td>
</tr>
<tr>
<td>median</td>
<td>Arithmetic median of non-NA values</td>
</tr>
<tr>
<td>std, var</td>
<td>Unbiased (n – 1 denominator) standard deviation and variance</td>
</tr>
<tr>
<td>min, max</td>
<td>Minimum and maximum of non-NA values</td>
</tr>
<tr>
<td>prod</td>
<td>Product of non-NA values</td>
</tr>
<tr>
<td>first, last</td>
<td>First and last non-NA values</td>
</tr>
</tbody>
</table>

[W. McKinney, Python for Data Analysis]
Iterating over groups

- for name, group in df.groupby('key1'):
  print(name)
  print(group)

- Can also .describe() groups
### Apply: Generalized methods

In [74]: def top(df, n=5, column='tip_pct'):
    ....:     return df.sort_values(by=column)[-n:]

In [75]: top(tips, n=5)
Out[75]:
            total_bill  tip smoker  day    time  size  tip_pct
109       14.31    4.00   Yes  Sat  Dinner    2  0.279525
183       23.17    6.50   Yes  Sun  Dinner    4  0.280535
232       11.61    3.39    No  Sat  Dinner    2  0.291990
  67       3.07     1.00   Yes  Sat  Dinner    1  0.325733
 178       9.60     4.00   Yes  Sun  Dinner    2  0.416667
 172       7.25     5.15   Yes  Sun  Dinner    2  0.710345

In [76]: tips.groupby('smoker').apply(top)
Out[76]:

<table>
<thead>
<tr>
<th>smoker</th>
<th>total_bill</th>
<th>tip smoker</th>
<th>day</th>
<th>time</th>
<th>size</th>
<th>tip_pct</th>
</tr>
</thead>
<tbody>
<tr>
<td>No</td>
<td>88</td>
<td>24.71</td>
<td>5.85</td>
<td>No</td>
<td>Thur</td>
<td>Lunch</td>
</tr>
<tr>
<td></td>
<td>185</td>
<td>20.69</td>
<td>5.00</td>
<td>No</td>
<td>Sun</td>
<td>Dinner</td>
</tr>
<tr>
<td></td>
<td>51</td>
<td>10.29</td>
<td>2.60</td>
<td>No</td>
<td>Sun</td>
<td>Dinner</td>
</tr>
<tr>
<td></td>
<td>149</td>
<td>7.51</td>
<td>2.00</td>
<td>No</td>
<td>Thur</td>
<td>Lunch</td>
</tr>
<tr>
<td></td>
<td>232</td>
<td>11.61</td>
<td>3.39</td>
<td>No</td>
<td>Sat</td>
<td>Dinner</td>
</tr>
<tr>
<td>Yes</td>
<td>109</td>
<td>14.31</td>
<td>4.00</td>
<td>Yes</td>
<td>Sat</td>
<td>Dinner</td>
</tr>
<tr>
<td></td>
<td>183</td>
<td>23.17</td>
<td>6.50</td>
<td>Yes</td>
<td>Sun</td>
<td>Dinner</td>
</tr>
<tr>
<td></td>
<td>67</td>
<td>3.07</td>
<td>1.00</td>
<td>Yes</td>
<td>Sat</td>
<td>Dinner</td>
</tr>
<tr>
<td></td>
<td>178</td>
<td>9.60</td>
<td>4.00</td>
<td>Yes</td>
<td>Sun</td>
<td>Dinner</td>
</tr>
<tr>
<td></td>
<td>172</td>
<td>7.25</td>
<td>5.15</td>
<td>Yes</td>
<td>Sun</td>
<td>Dinner</td>
</tr>
</tbody>
</table>
Apply

- `tips.groupby('smoker').apply(top)`
- Function is an **argument**
- Function applied on each **row group**
- All row groups glued together using `concat`
Types of GroupBy

• Aggregation: `agg`
  - `n:1` n group values become one value
  - Examples: mean, min, median

• Apply: `apply`
  - `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: `transform`
  - `n:n` n group values become n values
  - Cannot mutate the input
Transform Example

In Group Transforms and "Unwrapped" GroupBys

Here are the group means by key:

Let's consider a simple example for illustration:

In 12.2 Advanced GroupBy Use

While we've already discussed using the transformations. There is another built-in method called depth in Chapter 10

It must not mutate its input. It can produce an object of the same shape as the input group but imposes more constraints on the kind of function you can use:

•

•

•

...
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]:
   key  value
0    a     4.5
1    b     5.5
2    c     6.5
3    a     4.5
4    b     5.5
5    c     6.5
6    a     4.5
7    b     5.5
8    c     6.5
9    a     4.5
10   b     5.5
11   c     6.5
Name: value, dtype: float64

Or:
g.transform('mean')
Normalization

As a more complicated example, we can compute the ranks in descending order for

We can obtain equivalent results in this case either using

Consider a group transformation function composed from simple aggregations:

```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
```

==

While an unwrapped group operation may involve multiple group aggregations, the
overall benefit of vectorized operations often outweighs this.

(W. McKinney)
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>agffunc</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>
Pivot Tables in Pandas

• **tips**

```python
In [26]:
In [29]:
In [28]:
In [30]:
Out[26]:
smoker
No     count    151.000000
       mean       0.159328
       std        0.039910
       min        0.056797
       25%        0.136906
       50%        0.155625
       75%        0.185014
       max        0.291990
Yes     count     93.000000
       mean       0.163196
       std        0.085119
       min        0.035638
       25%        0.106771
       50%        0.153846
       75%        0.195059
       max        0.710345
Name: tip_pct, dtype: float64
Out[29]:
count  mean  std  min  25%  50%  75%  max
smoker
No     151.0 0.159328 0.039910 0.056797 0.136906 0.155625 0.185014 0.291990
Yes    93.0 0.163196 0.085119 0.035638 0.106771 0.153846 0.195059 0.710345
Out[28]:
smoker
No     0.206140
Yes    0.236398
Name: tip_pct, dtype: float64
Out[30]:
size  tip  tip_pct  total_bill
sex  smoker
Female No      2.592593 2.773519 0.156921 18.105185
     Yes     2.242424 2.931515 0.182150 17.977879
Male  No      2.711340 3.113402 0.160669 19.791237
     Yes     2.500000 3.051167 0.152771 22.284500
```

• **tips.pivot_table(index=['sex', 'smoker'])**
Pivot Tables with Margins and Aggfunc

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

<table>
<thead>
<tr>
<th></th>
<th>sex</th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>day</td>
<td>smoker</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>No</td>
<td>Yes</td>
</tr>
<tr>
<td>Female</td>
<td>Fri</td>
<td>2.0</td>
<td>7.0</td>
</tr>
<tr>
<td></td>
<td>Sat</td>
<td>13.0</td>
<td>15.0</td>
</tr>
<tr>
<td></td>
<td>Sun</td>
<td>14.0</td>
<td>4.0</td>
</tr>
<tr>
<td></td>
<td>Thur</td>
<td>25.0</td>
<td>7.0</td>
</tr>
<tr>
<td>Male</td>
<td>Fri</td>
<td>2.0</td>
<td>8.0</td>
</tr>
<tr>
<td></td>
<td>Sat</td>
<td>32.0</td>
<td>27.0</td>
</tr>
<tr>
<td></td>
<td>Sun</td>
<td>43.0</td>
<td>15.0</td>
</tr>
<tr>
<td></td>
<td>Thur</td>
<td>20.0</td>
<td>10.0</td>
</tr>
<tr>
<td>All</td>
<td></td>
<td>151.0</td>
<td>93.0</td>
</tr>
</tbody>
</table>
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
### Crosstabs

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

<table>
<thead>
<tr>
<th></th>
<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>
<td></td>
</tr>
<tr>
<td>Dinner</td>
<td>Fri</td>
<td>3</td>
<td>9</td>
<td>12</td>
</tr>
<tr>
<td></td>
<td>Sat</td>
<td>45</td>
<td>42</td>
<td>87</td>
</tr>
<tr>
<td></td>
<td>Sun</td>
<td>57</td>
<td>19</td>
<td>76</td>
</tr>
<tr>
<td></td>
<td>Thur</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>Lunch</td>
<td>Fri</td>
<td>1</td>
<td>6</td>
<td>7</td>
</tr>
<tr>
<td></td>
<td>Thur</td>
<td>44</td>
<td>17</td>
<td>61</td>
</tr>
<tr>
<td><strong>All</strong></td>
<td></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)`
Time Series Data
What is time series data?

• Technically, it's normal tabular data with a timestamp attached
• But… we have observations of the same values over time, usually in order
• This allows more analysis
• Example: Web site database that tracks the last time a user logged in
  - 1: Keep an attribute `lastLogin` that is overwritten every time user logs in
  - 2: **Add a new row** with login information every time the user logs in
  - Option 2 takes more storage, but we can also do a lot more analysis!
Time Series Data

- Metrics: measurements at regular intervals
- Events: measurements that are not gathered at regular intervals
Types of Time Series Data

• time series: observations for a single entity at different time intervals
  - one patient's heart rate every minute

• cross-section: observations for multiple entities at the same point in time
  - heart rates of 100 patients at 8:01pm

• panel data: observations for multiple entities at different time intervals
  - heart rates of 100 patients every minute over the past hour
Time Series Databases

- Most time series data is heavy **inserts**, few updates
- Also analysis tends to be on ordered data with trends, prediction, etc.
- Can also consider **stream** processing
- Focus on time series allows databases to specialize
- Examples:
  - InfluxDB (noSQL)
  - TimescaleDB (SQL-based)
Features of Time Series Data

- Trend: long-term increase or decrease in the data
- Seasonal Pattern: time series is affected by seasonal factors such as the time of the year or the day of the week (fixed and of known frequency)
- Cyclic Pattern: rises and falls that are not of a fixed frequency
- Stationary: no predictable patterns (roughly horizontal with constant variance)
  - White noise series is stationary
  - Will look the basically the same whenever you observe it
Examples

- **US Treasury bill contracts**
- **Australian electricity production**
- **Sales of new one-family houses, USA**
- **Annual Canadian Lynx trappings**

[References: R. J. Hyndman]
Examples

Trend

US Treasury bill contracts

Australian electricity production

Sales of new one-family houses, USA

Annual Canadian Lynx trappings

D. Koop, CSCI 680/490, Spring 2022
Examples

- **US Treasury bill contracts**
  - Trend

- **Australian electricity production**
  - Trend + Seasonality

- **Sales of new one–family houses, USA**

- **Annual Canadian Lynx trappings**

[R. J. Hyndman]
Examples

Trend

Seasonality + Cyclic

Trend + Seasonality

[R. J. Hyndman]
Examples

US Treasury bill contracts

Australian electricity production

Sales of new one-family houses, USA

Annual Canadian Lynx trappings

Trend

Seasonality + Cyclic

Trend + Seasonality

Stationary

[R. J. Hyndman]
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>)`
Datetime format specification

• Look it up:
  - http://strftime.org

• Generally, can create whatever format you need using these format strings
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
More Pandas Support

• Accessing a particular time or checking equivalence allows any string that can be interpreted as a date:
  - `ts['1/10/2011']` or `ts['20110110']`

• Date ranges: `pd.date_range('4/1/2012','6/1/2012',freq='4h')`

• 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'

In [84]: pd.date_range('5/2/2012 12:56:31', periods=5, normalize=True)

Out[84]:
<class 'pandas.tseries.index.DatetimeIndex'>
[2012-05-02, ..., 2012-05-06]
Length: 5, Freq: D, Timezone: None

Frequencies and Date Offsets

In [85]: from pandas.tseries.offsets import Hour, Minute
In [86]: hour = Hour()
In [87]: hour
Out[87]: <Hour>

In [88]: four_hours = Hour(4)
In [89]: four_hours
Out[89]: <4 * Hours>

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

In [91]: Hour(2) + Minute(30)

Out[91]: <150 * Minutes>

In [92]: pd.date_range('1/1/2000', periods=10, freq='1h30min')

Out[92]:
<class 'pandas.tseries.index.DatetimeIndex'>
[2000-01-01 00:00:00, ..., 2000-01-01 13:30:00]
Length: 10, Freq: 90T, Timezone: None
### 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.</td>
</tr>
</tbody>
</table>
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 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   -0.066748
2000-02-29    0.838639
2000-03-31   -0.117388
2000-04-30   -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())`
  
<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
- `diff = parse_date("1 Jan 2017") - datetime.now().date()`
- `diff.days`

- Also a `pd.Timedelta` object that take strings:
  - `datetime.now().date() + pd.Timedelta("4 days")`

- Also, Roll dates using anchored offsets
  - `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')`