Examples

Traces aims to make it simple to write *readable code* to:

- **Munge**. Read, write, and manipulate unevenly-spaced time series data
- **Explore**. Perform basic analyses of unevenly-spaced time series data without making an awkward / lossy transformation to evenly-spaced representations
- **GTFO**. Gracefully transform unevenly-spaced times series data to evenly-spaced representations

This section has a few examples of how to do these things.

Read and manipulate

Say we have a directory with a bunch of CSV files with information about light bulbs in a home. Each CSV file has the wattage used by the bulb as a function of time. Some of the light bulbs only send a signal when the state changes, but some send a signal every minute. We can read them with this code.

```
def parse_iso_datetime(value):
    return datetime.strptime(value, "%Y-%m-%dT%H:%M:%S")


def read_all(pattern='data/lightbulb-*.csv'):
    """Read all of the CSVs in a directory matching the filename pattern
    as TimeSeries.

    """
    result = []
    for filename in glob.iglob(pattern):
        print('reading', filename, file=sys.stderr)
        ts = traces.TimeSeries.from_csv(
            filename,
            time_column=0,
            time_transform=parse_iso_datetime,
            value_column=1,
            value_transform=int,
            default=0,
        )
        ts.compact()
        result.append(ts)
    return result


ts_list = read_all()
```

The call to `ts.compact()` will remove any redundant measurements. Depending on how often your data changes compared to how often it is sampled, this can reduce the size of the data dramatically.

Basic analysis

Now, let's say we want to do some basic exploratory analysis of how much power is used in the whole home. We'll first take all of the individual traces and merge the v: latest a single TimeSeries where the values is the total wattage.

```
total_watts = traces.TimeSeries.merge(ts_list, operation=sum)
```

The merged time series has times that are the union of all times in the individual series. Since each time series is the wattage of the lightbulb, the values after the sum are the total wattage used over time. Here's how to check the mean power consumption in January.

```
histogram = total_watts.distribution(
    start=datetime(2016, 1, 1),
    end=datetime(2016, 2, 1),
)
print(histogram.mean())
```

Let's say we want to break this down to see how the distribution of power consumption varies by time of day.

```
for hour, distribution in total_watts.distribution_by_hour_of_day():
    print(hour, distribution.quantiles([0.25, 0.5, 0.75]))
```

Or day of week.

```
for day, distribution in total_watts.distribution_by_day_of_week():
    print(day, distribution.quantiles([0.25, 0.5, 0.75]))
```

Finally, we just want to look at the distribution of power consumption during business hours on each day in January.

```
for t in datetime_range(datetime(2016, 1, 1), datetime(2016, 2, 1), 'days'):
    biz_start = t + timedelta(hours=8)
    biz_end = t + timedelta(hours=18)
    histogram = total_watts.distribution(start=biz_start, end=biz_end)
    print(t, histogram.quantiles([0.25, 0.5, 0.75]))
```

In practice, you'd probably be plotting these distribution and time series using your tool of choice.

Transform to evenly-spaced

Now, let's say we want to do some forecasting of the power consumption of this home. There is probably some seasonality that need to be accounted for, among other things, and we know that statsmodels and pandas are tools with some batteries included for that type of thing. Let's convert to a pandas Series.

```
regular = total_watts.moving_average(300, pandas=True)
```

That will convert to a regularly-spaced time series using a moving average to avoid aliasing (more info here). At this point, a good next step is the excellent tutorial by Tom Augspurger, starting with the *Modeling Time Series* section.

v: latest