Timetable Documentation

Release 0.2

Ontje Lünsdorf

Sep 15, 2017

Contents

iCal is a relatively compact albeit unintuitive and inconvenient text format for calendaring and scheduling information. Recurring events can be defined through a large set of rules, which makes the format compact but inconvenient for automatic processing. The intention of this package is to provide functions to ease the automatic processing of iCalendar files by converting the iCalendar information into a timetable.

CHAPTER 1

Timetable data

A timetable is a sorted sequence of tuples containing start and end datetime of an entry, for example `(start, end, entry)`. `start` and `end` are `datetime` offset-naive objects (e.g. containing no timezone information) in UTC time. The `entry` is a dictionary, containing arbitrary values.

The following example is a valid timetable list:

```
[
    (datetime(2015, 1, 1, 12), datetime(2015, 1, 1, 13), {}),
    (datetime(2015, 1, 2, 12), datetime(2015, 1, 2, 13), {}),
    (datetime(2015, 1, 3, 12), datetime(2015, 1, 3, 13), {}),
]
```

Timetables can be generated from iCal files. The following example shows how a timetable is generated from all VEVENT entries in a iCal calendar. The example prints the start `datetime` of each entry as well as the calendar events summary:

```
>>> from timetable import parse_ical, generate_timetable
>>>
>>> icaldata = b"""
... BEGIN:VCALENDAR
... BEGIN:VEVENT
... UID:0
... DTSTART:20150101T120000Z
... DTEND:20150101T130000Z
... RRULE:FREQ=DAILY;COUNT=3;BYDAY=TH,FR
... SUMMARY:event a
... END:VEVENT
... BEGIN:VEVENT
... UID:1
... DTSTART:20150101T123000Z
... DTEND:20150101T133000Z
... RRULE:FREQ=DAILY;COUNT=3
... SUMMARY:event b
... END:VEVENT
... END:VCALENDAR
```

```
... """
>>> calendar = parse_ical(icaldata)[0]
>>> for start, end, entry in generate_timetable(calendar, b'vevent'):
...     print('%s %s' % (start.isoformat(),
...             str(entry['item'][b'summary'][0].value.decode('utf-8'))))
2015-01-01T12:00:00 event a
2015-01-01T12:30:00 event b
2015-01-02T12:00:00 event a
2015-01-02T12:30:00 event b
2015-01-03T12:30:00 event b
2015-01-08T12:00:00 event a
```

CHAPTER 2

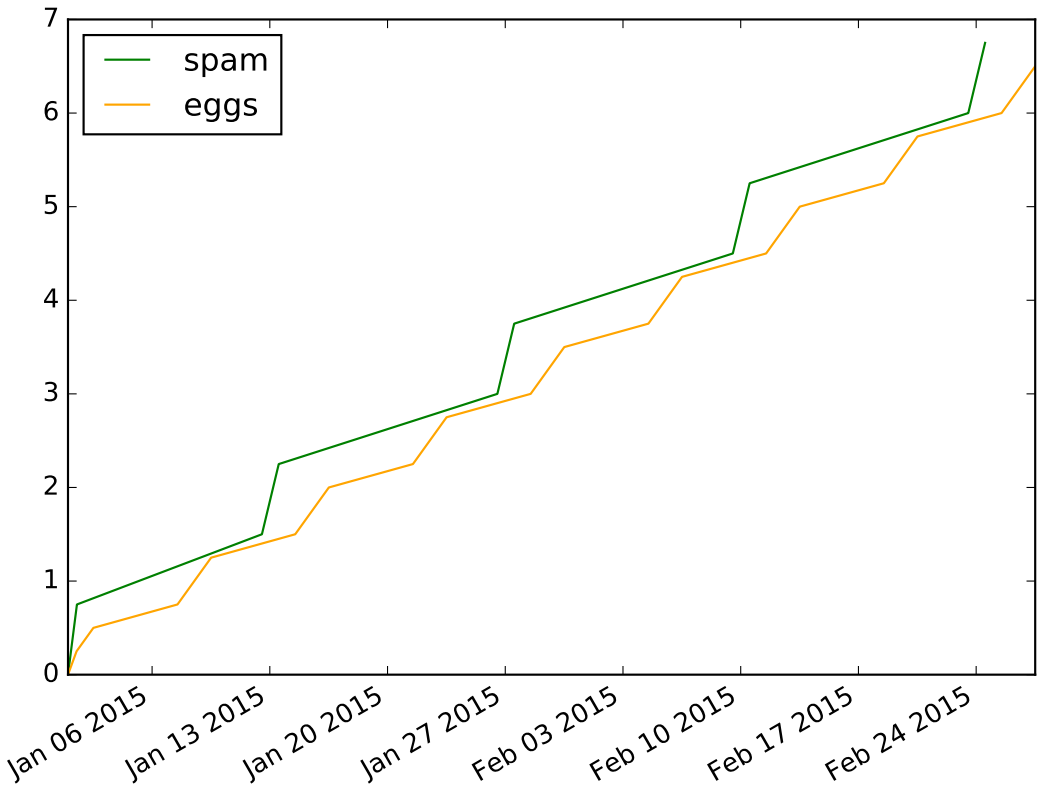
Example application

Timetable data can be used to calculate metrics for time management, like for example, compute the time spent in meetings or working on projects.

However, of more pressing concern is the question if you spent more time eating eggs or spam for lunch in two months, assuming you eat spam for 45 minutes but only every other week on monday and tuesday whereas you eat eggs for 15 minutes every week but only from wednesday to friday:

```
>>> from matplotlib import pyplot as plt
>>> from datetime import datetime, timedelta
>>> from timetable import parse_ical, generate_timetable, clip_timetable
>>>
>>> icaldata = b"""
... BEGIN:VCALENDAR
... BEGIN:VEVENT
... UID:0
... DTSTART:20150101T120000Z
... DTEND:20150101T124500Z
... RRULE:FREQ=WEEKLY;BYDAY=MO,TU;INTERVAL=2
... SUMMARY:spam
... END:VEVENT
... BEGIN:VEVENT
... UID:1
... DTSTART:20150101T120000Z
... DTEND:20150101T121500Z
... RRULE:FREQ=WEEKLY;BYDAY=WE,TH,FR
... SUMMARY:eggs
... END:VEVENT
... END:VCALENDAR
... """
>>> calendar = parse_ical(icaldata)[0]
>>>
>>> start = datetime(2015, 1, 1)
>>> end = datetime(2015, 3, 1)
>>> timetable = clip_timetable(generate_timetable(calendar), start, end)
>>>
```

```
>>> dates = {b'spam': [start], b'eggs': [start]}
>>> cumulative = {b'spam': [0], b'eggs': [0]}
>>> for start, end, entry in timetable:
...     summary = entry['item'][b'summary'][0].value
...     time = (end - start).total_seconds() / 3600
...     dates[summary].append(end)
...     cumulative[summary].append(cumulative[summary][-1] + time)
>>>
>>> plt.plot(dates[b'spam'], cumulative[b'spam'], color='green', label='spam')
[...]
>>> plt.plot(dates[b'eggs'], cumulative[b'eggs'], color='orange', label='eggs')
[...]
>>> plt.legend(loc='upper left')
<...>
>>> plt.gcf().autofmt_xdate()
>>> plt.show()
```



It is spam.

CHAPTER 3

Contents

Guide

This guide should introduce to the functionality provided in timetable. The guide uses two example calendar files, one containing work appointments and the other one containing duty information (e.g. work days, vacation and holidays).

A cumulative plot - in detail

Let's start with an extensive example to explain the low-level primitives of timetable. The objective is to build a cumulative plot of the time spent during work. To spice things up, lets say that we want to get a daily overview, e.g. summing up the daily work.

The first step is to load the calendar file:

```
>>> from datetime import datetime, time, timedelta
>>> from timetable import (load_timetable, cut_timetable,
...         merge_intersections, annotate_timetable, collect_keys)
>>>
>>> start, end = datetime(2015, 1, 1), datetime(2015, 2, 1)
>>> calendar_files = {'Work': open('data/work.ics', 'rb')}
>>> timetable = load_timetable(calendar_files, clip_start=start, clip_end=end)
```

The *load_timetable()* function supports loading and merging multiple calendars at once, which is why a dictionary must be passed in containing the calendar files. Furthermore, the function also annotates entries with tags. If a calendar entry does not contain a tag, the key of the dictionary is used by default.

Note: It is a good idea to supply at least a *clip_end* date. Otherwise, an infinite amount of entries might get generated, if there are unlimited recurring events.

```
>>> days = []
>>> day = start
```

```
>>> while day < end:
...     days.append(datetime.combine(day, time()))
...     day += timedelta(days=1)
```

The next step is to compute a list of `datetime` objects at which we want to cut the timetable entries.

```
>>> # Cut timetable entries at day boundaries.
>>> work = {}
>>> for idx, day_timetable in enumerate(cut_timetable(timetable, days)):
...     # Merge all intersecting entries.
...     day_timetable = merge_intersections(day_timetable)
...     # Collect tags from merged entries.
...     day_timetable = annotate_timetable(day_timetable, collect_keys('tags'))
...     for start, end, entry in day_timetable:
...         # Compute duration and apply to timeseries.
...         duration = (end - start).total_seconds() / len(entry['tags'])
...         for tag in entry['tags']:
...             if not tag in work:
...                 work[tag] = [0 for day in days]
...             work[tag][idx] += duration
```

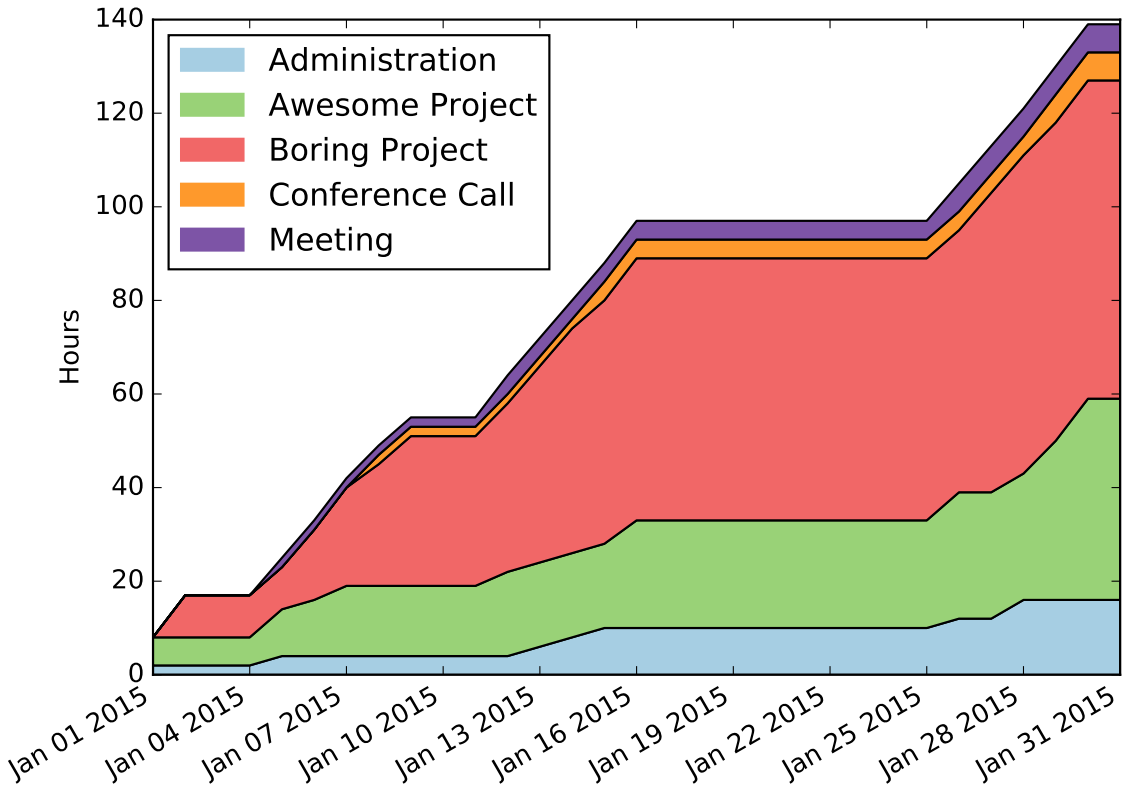
The separation of a timetable into daily timetables is done by the function *cut_timetable()*.

Entries of a daily timetable might overlap. The *merge_intersections()* function merges entries from a timetable based on a key and generates non-intersecting entries.

The duration of an entry may now be calculated by simply subtracting the start datetime from the end datetime. The duration is accounted equally for each tag.

```
>>> import numpy as np
>>> import matplotlib.pyplot as plt
>>> from matplotlib.patches import Patch
>>> from matplotlib.cm import ScalarMappable
>>>
>>> fig = plt.figure()
>>> ax = fig.add_subplot(1, 1, 1)
>>>
>>> # Draw a stackplot with custom colors and legend handles.
>>> colormap = ScalarMappable(cmap='Paired')
>>> colormap.set_clim(0, len(work))
>>> data, colors, handles, labels = [], [], [], []
>>> for idx, name in enumerate(sorted(work)):
...     color = colormap.to_rgba(idx)
...     data.append(np.cumsum(work[name]) / 3600)
...     colors.append(color)
...     handles.append(Patch(color=color))
...     labels.append(name)
>>> ax.stackplot(days, data, colors=colors)
[...]
>>>
>>> fig.autofmt_xdate()
>>> ax.set_ylabel('Hours')
<...>
>>> ax.legend(handles, labels, loc='upper left')
<...>
>>>
>>> plt.show()
```

The last code fragment builds a stacked cumulative plot for each of the collected time series.



Using convenience functions

Timetable provides convenience functions for some of the tasks detailed in the above example. The following code computes the same `work` data as in the example:

```
>>> from datetime import datetime, timedelta
>>> from timetable import load_timetable, sum_timetable, datetime_range
>>>
>>> start, end = datetime(2015, 1, 1), datetime(2015, 2, 1)
>>> days = list(datetime_range(start, end, timedelta(days=1)))
>>>
>>> work = sum_timetable(load_timetable({'work': open('data/work.ics', 'rb')},
...         clip_start=start, clip_end=end), days)
```

The function *`datetime_range()`* is used to generate a list of `datetime` objects. `sum_timetable()` computes the activity timeseries data of group by a given key (e.g. the tags).

Adding duty information

Lets add duty information to the visualization. This example shows how to do basic timeseries arithmetic. The duty calendar contains a recurring event for the work duty (e.g. recurring each workday for a duration of 8 hours). Holidays and vacations are additional events and need to be subtracted from the work duty.

```
>>> from datetime import datetime, timedelta
>>> from timetable import load_timetable, sum_timetable, datetime_range
>>>
>>> start, end = datetime(2015, 1, 1), datetime(2015, 2, 1)
>>> days = list(datetime_range(start, end, timedelta(days=1)))
>>>
>>> work = sum_timetable(load_timetable({'Work': open('data/work.ics', 'rb')},
...          clip_start=start, clip_end=end), days)
>>> duty = sum_timetable(load_timetable({'Duty': open('data/duty.ics', 'rb')},
...          clip_start=start, clip_end=end), days)
```

We start by loading each calendar in two timetables and summing them up on each day.

```
>>> duty_series = [max(w - h - v, 0) / 3600
...          for w, h, v in zip(duty['Duty'], duty['Holiday'], duty['Vacation'])]
```

Now we need to subtract the holiday and vacation time from the duty time.

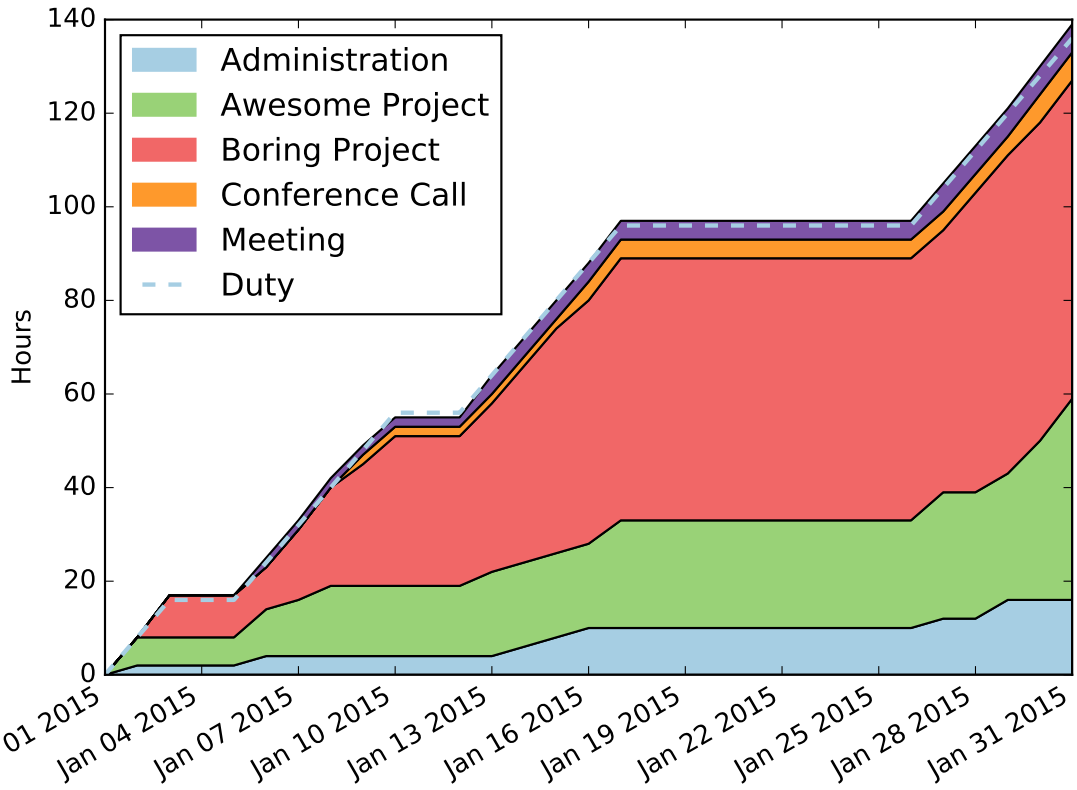
```
>>> import numpy as np
>>> import matplotlib.pyplot as plt
>>> from matplotlib.patches import Patch
>>> from matplotlib.cm import ScalarMappable
>>>
>>> fig = plt.figure()
>>> ax = fig.add_subplot(1, 1, 1)
>>>
>>> # Draw a stackplot with custom colors and legend handles.
>>> colormap = ScalarMappable(cmap='Paired')
>>> colormap.set_clim(0, len(work))
>>> data, colors, handles, labels = [], [], [], []
>>> for idx, name in enumerate(sorted(work)):
...     color = colormap.to_rgba(idx)
...     data.append(np.cumsum(work[name]) / 3600)
...     colors.append(color)
...     handles.append(Patch(color=color))
...     labels.append(name)
>>> ax.stackplot(days, data, colors=colors)
[...]
```

Draw the stacked cumulative plot as above.

```
>>> handles.extend(ax.plot(days, np.cumsum(duty_series), lw=2, ls='--'))
>>> labels.append('Duty')
```

Add a line for the duty time series.

```
>>> fig.autofmt_xdate()
>>> ax.set_ylabel('Hours')
<...>
>>> ax.legend(handles, labels, loc='upper left')
<...>
>>>
>>> plt.show()
```



There is a vacation in the third week of january in the example calendar file, which is also visible in the cumulative duty time.

Examples

The following examples should help getting insight into timetables.

Cutting a timetable

The *cut_timetable()* cuts a timetable at specific cut points and returns a sub-timetable for each cut interval. The following example shows how a timetable with an entry spanning one month is cut in weekly intervals:

```
>>> from datetime import datetime, date, timedelta
>>> from timetable import cut_timetable, datetime_range
>>>
>>> timetable = [
...     (datetime(2015, 1, 1), datetime(2015, 2, 1), {'name': 'spam'}),
... ]
>>> cuts = datetime_range(datetime(2015, 1, 1), datetime(2015, 2, 1),
...         timedelta(days=7))
>>> for sub_timetable in cut_timetable(timetable, cuts):
...     for start, end, entry in sub_timetable:
```

```
...         print('%s %s %s' % (start.isoformat(), end.isoformat(), entry))
2015-01-01T00:00:00 2015-01-08T00:00:00 {'name': 'spam'}
2015-01-08T00:00:00 2015-01-15T00:00:00 {'name': 'spam'}
2015-01-15T00:00:00 2015-01-22T00:00:00 {'name': 'spam'}
2015-01-22T00:00:00 2015-01-29T00:00:00 {'name': 'spam'}
```

Annotating a timetable

A timetable consists of `(start, end, entry)` tuples, where *start* and *end* are `datetime` objects and *entry* is a dictionary. The *entry* dictionary contains arbitrary keys and values. You can add additional keys to this dictionary using the *`annotate_timetable()`* function. The following example shows how to compute the hour duration for each entry and add the result under the key *hours* to the entry.

```
>>> from datetime import datetime, date, timedelta
>>> from timetable import cut_timetable, annotate_timetable, datetime_range
>>>
>>> timetable = [
...     (datetime(2015, 1, 1), datetime(2015, 2, 1), {'name': 'spam'}),
... ]
>>> cuts = datetime_range(datetime(2015, 1, 1), datetime(2015, 2, 1),
...         timedelta(days=7))
>>>
>>> def calc_hours(start, end, entry):
...     entry['hours'] = (end - start).total_seconds() / 3600
>>>
>>> for sub_timetable in cut_timetable(timetable, cuts):
...     for start, end, entry in annotate_timetable(sub_timetable, calc_hours):
...         print('%s %s %s' % (start.isoformat(), end.isoformat(),
...                 entry['hours']))
2015-01-01T00:00:00 2015-01-08T00:00:00 168.0
2015-01-08T00:00:00 2015-01-15T00:00:00 168.0
2015-01-15T00:00:00 2015-01-22T00:00:00 168.0
2015-01-22T00:00:00 2015-01-29T00:00:00 168.0
```

Merging timetables

If you need to work with multiple timetables (for example from multiple calendars), you can use the *`merge_timetables()`* function to merge them into a single timetable.

```
>>> from datetime import datetime
>>> from timetable import merge_timetables
>>>
>>> timetable_spam = [
...     (datetime(2015, 1, 1), datetime(2015, 1, 2), {'name': 'spam'}),
...     (datetime(2015, 1, 3), datetime(2015, 1, 5), {'name': 'spam'}),
... ]
>>> timetable_eggs = [
...     (datetime(2015, 1, 2), datetime(2015, 1, 3), {'name': 'eggs'}),
...     (datetime(2015, 1, 4), datetime(2015, 1, 5), {'name': 'eggs'}),
... ]
>>>
>>> for start, end, entry in merge_timetables([timetable_spam, timetable_eggs]):
...     print('%s %s %s' % (start.isoformat(), end.isoformat(), entry))
2015-01-01T00:00:00 2015-01-02T00:00:00 {'name': 'spam'}
```

```
2015-01-02T00:00:00 2015-01-03T00:00:00 {'name': 'eggs'}
2015-01-03T00:00:00 2015-01-05T00:00:00 {'name': 'spam'}
2015-01-04T00:00:00 2015-01-05T00:00:00 {'name': 'eggs'}
```

Merge intersections

Timetable entries might overlap with each other. The function *merge_intersections()* merges overlapping entries, thereby generating a non-overlapping timetables. The entries of the non-overlapping timetable contain a single key `entries`, whose value is the list of merged entries.

```
>>> from datetime import datetime
>>> from timetable import merge_intersections
>>>
>>> timetable = [
...     (datetime(2015, 1, 1), datetime(2015, 1, 3), {'name': 'spam'}),
...     (datetime(2015, 1, 2), datetime(2015, 1, 5), {'name': 'eggs'}),
...     (datetime(2015, 1, 4), datetime(2015, 1, 6), {'name': 'spam'}),
... ]
>>>
>>> for start, end, entry in merge_intersections(timetable):
...     print('%s %s %s' % (start.isoformat(), end.isoformat(), entry))
2015-01-01T00:00:00 2015-01-02T00:00:00 {'entries': [{'name': 'spam'}]}
2015-01-02T00:00:00 2015-01-03T00:00:00 {'entries': [{'name': 'spam'}, {'name': 'eggs
↪'}]}
2015-01-03T00:00:00 2015-01-04T00:00:00 {'entries': [{'name': 'eggs'}]}
2015-01-04T00:00:00 2015-01-05T00:00:00 {'entries': [{'name': 'eggs'}, {'name': 'spam
↪'}]}
2015-01-05T00:00:00 2015-01-06T00:00:00 {'entries': [{'name': 'spam'}]}
```

Summing it up (literally)

The following example puts all the above pieces together to compute the duration of all entries with a given key. The duration is computed using another annotation function *compute_duration()*. This function distributes the duration equally to each key in case of overlaps.

```
>>> from datetime import datetime
>>> from timetable import (merge_intersections, annotate_timetable,
...         collect_keys, compute_duration)
>>>
>>> timetable = [
...     (datetime(2015, 1, 1), datetime(2015, 1, 3), {'name': 'spam'}),
...     (datetime(2015, 1, 2), datetime(2015, 1, 5), {'name': 'eggs'}),
...     (datetime(2015, 1, 4), datetime(2015, 1, 6), {'name': 'spam'}),
... ]
>>>
>>> timetable = merge_intersections(timetable)
>>> timetable = annotate_timetable(timetable, collect_keys(key='name'))
>>> timetable = annotate_timetable(timetable, compute_duration(key='name'))
>>>
>>> # Sum durations for each key.
>>> durations = {}
>>> for start, end, entry in timetable:
...     for name in entry['name']:
...         if not name in durations:
```

```
...             durations[name] = 0
...         durations[name] += entry['duration']
>>>
>>> for name in sorted(durations):
...     print('%s %s' % (name, durations[name] / 3600))
eggs 48.0
spam 72.0
```

There's also a convenience function *sum_timetable()* available, that does all these steps at once. In addition, this function also cuts the timetable, thereby generating a series of durations.

```
>>> from datetime import datetime
>>> from timetable import sum_timetable, datetime_range
>>>
>>> timetable = [
...     (datetime(2015, 1, 1), datetime(2015, 1, 3), {'name': 'spam'}),
...     (datetime(2015, 1, 2), datetime(2015, 1, 5), {'name': 'eggs'}),
...     (datetime(2015, 1, 4), datetime(2015, 1, 6), {'name': 'spam'}),
... ]
>>>
>>> # Compute total durations.
>>> durations = sum_timetable(timetable,
...         cuts=[datetime(2015, 1, 1), datetime(2015, 1, 6)], key='name')
>>> for name in sorted(durations):
...     print('%s %s' % (name, [d / 3600. for d in durations[name]]))
eggs [0.0, 48.0]
spam [0.0, 72.0]
>>>
>>> # Compute daily durations.
>>> cuts = datetime_range(datetime(2015, 1, 1), datetime(2015, 1, 6),
...         timedelta(days=1))
>>> durations = sum_timetable(timetable, cuts=cuts, key='name')
>>> for name in sorted(durations):
...     print('%s %s' % (name, [d / 3600. for d in durations[name]]))
eggs [0.0, 0.0, 12.0, 24.0, 12.0]
spam [0.0, 24.0, 12.0, 0.0, 12.0]
```

API

timetable.timetable

Functions for timetable modification, for example merging, tagging and cutting at specific dates.

timetable.timetable.**annotate_tags**(*tag_pat=re.compile('\\[([^]]*)\\] *')*, *emptytag='misc'*)

Returns an annotation function which parses the items summary for a tag. A tag is identified through the regular expression *tag_pat*. If no tag is found, *emptytag* is applied.

Note: The summary is assumed to be encoded as UTF-8.

---

timetable.timetable.**annotate_timetable**(*timetable*, **annotate_funcs*)

Annotates all entries of *timetable* with the result of all *annotate_funcs*. The annotation functions must accept the arguments *start*, *end*, *entry*.

timetable.timetable.**clip_timetable**(*timetable*, *clip_start=None*, *clip_end=None*, *pending=None*)
Generates a timetable by clipping entries from the given *timetable*. Entries ending before *clip_start* are discarded as well as entries starting after *clip_end*. Start and end times of entries lying on the boundaries modified to match *clip_start* resp. *clip_end*. Entries on the *clip_end* are added to the list *pending*, if it is supplied.

timetable.timetable.**collect_keys**(*key='tags'*, *collection='entries'*)
Returns an annotation function which collects *key* from a *collection*. The resulting set of keys is added to the entries dictionary under the key *key*.

This function is useful to extract tags from a merged timetable as returned by *merge_intersections()* for example.

timetable.timetable.**compute_duration**(*key='tags'*)
Returns an annotation function which computes the duration of an entry. The duration is allocated equally for each *key* of the entry (e.g. divided by the amount of keys).

timetable.timetable.**cut_timetable**(*timetable*, *cuts=(None*, *None)*)
Generates a timetable by cutting entries of *timetable* at the given *cuts* datetimes.

timetable.timetable.**generate_timetable**(*calendar*, *itemtype=b'vevent'*)
Generates a timetable from all items of type *itemtype* in the given *calendar*.

timetable.timetable.**load_timetable**(*calendar_data*, *clip_start*, *clip_end*, *tag_pat=re.compile('\\[([^]]*)\\] *')*)
Loads and tags all events from the calendar files in the *calendar_files* dictionary. The keys in *calendar_files* are passed into *annotate_tags()* as emptytag. The values in *calendar_files* are filenames to iCal calendars. All events are clipped to *clip_start* and *clip_end*.

timetable.timetable.**merge_intersection**(*entries*, *start*, *end*)
Generates a non-overlapping timetable from *entries*. *start* and *end* limit the timespan for the intersection generation. The resulting timetable contains entry dictionaries with the single key `entries`, whose value is the list of merged entries.

timetable.timetable.**merge_intersections**(*timetable*)
Generates a timetable with merged intersection of entries in *timetable*. The resulting timetable will only contain entries with a single key `entries`, whose value is the list of the merged entries.

timetable.timetable.**merge_timetables**(*timetables*)
Generates a merged timetable from the given *timetables*. Entries are sorted by their start time.

timetable.timetable.**sum_timetable**(*timetable*, *cuts*, *key='tags'*)
Computes a dictionary with timeseries of the activity for each *key* in the given *cuts*. The activity duration is distributed evenly if there are intersections.

timetable.event

Functions for generating timetables from raw calendar components. See section *Timetable data* for an explanation of timetables.

timetable.event.**generate_item_timetable**(*uid*, *group*, *timezones*)
Generates a timetable for a calendar item. The item is given by its *uid* and a list of calendar components in *group*. It is required to supply the *timezones* of the calendar.

timetable.event.**uidgroups_by_type**(*calendar*, *itemtype*)
Selects all items of type *itemtype* in the *calendar* and groups them into a dictionary based on their UID.

timetable.ical

Low-level functions to parse iCal data into an object representation.

class timetable.ical.**Entry**(*name*, *attrs*, *value*)
Represents an iCal entry.

class timetable.ical.**Item**(*type=None*)
Represents an iCal item.

timetable.ical.**parse_ical**(*icalfile*)
Parses the *icalfile* and returns a list of *Item*. *icalfile* may be a str or a file-like object.

timetable.util

Utility functions.

timetable.util.**datetime_range**(*start*, *end*, *step*)
Generates dates from *start* (inclusive) to *end* (exclusive) with the given *step*.

The API is consists of separate layers.

The upper layer supplies convenience functions for handling timetables, for example merging and summarizing of timetables. The implementation is found in *timetable.timetable*:

generate_timetable	Generates a timetable from all items of type *itemtype* in the given *calendar*.
annotate_timetable	Annotates all entries of *timetable* with the result of all *annotate_funcs*.
cut_timetable	Generates a timetable by cutting entries of *timetable* at the given *cuts* datetimes.
merge_intersections	Generates a timetable with merged intersection of entries in *timetable*.
merge_timetables	Generates a merged timetable from the given *timetables*.
sum_timetable	Computes a dictionary with timeseries of the activity for each *key* in the given *cuts*.

The middle layer handles grouping of calendar items as well as evaluating recurrences and timezones. It is implemented in *timetable.event*:

uidgroups_by_type	Selects all items of type *itemtype* in the *calendar* and groups them into a dictionary based on their UID.
generate_item_timetable	Generates a timetable for a calendar item.

The lowest layer supplies functions for parsing iCal data and is implemented in the module *timetable.ical*:

Item	Represents an iCal item.
Entry	Represents an iCal entry.
parse_ical	Parses the *icalfile* and returns a list of *Item*.

Furthermore, there is also a small utility module *timetable.util* available:

datetime_range	Generates dates from *start* (inclusive) to *end* (exclusive) with the given *step*.

CHAPTER 4

Indices and tables

- genindex
- modindex
- search

Python Module Index

t

Index