# Python-Tempo Documentation

***Release 0.1.0***

**Andrew Pashkin**

October 25, 2015

Contents


**1 Links** **3**

**2 Features** **5**

**3 Quick example** **7**

**4 Schedule model** **9**
4.1 Example . . . . . 9
4.2 Informal definition . . . . . 9

**5 Alternatives** **11**

**6 TODO** **13**

**7 Documentation** **15**
7.1 Prerequisites . . . . . 15
7.2 Installation . . . . . 15
7.3 Usage . . . . . 16
7.4 API reference . . . . . 19

**8 Indices and tables** **25**

**Python Module Index** **27**

This project provides a generic way to compose and query schedules of recurrent continuous events, such as working time of organizations, meetings, movie shows, etc.

It contains a Python implementation and bindings for PostgreSQL, Django and Django REST Framework.

CHAPTER 1

# Links

**PyPI** https://pypi.python.org/pypi/python-tempo

**Documentation** https://python-tempo.readthedocs.org/

**Issues** https://github.com/AndrewPashkin/python-tempo/issues/

**Code** https://github.com/AndrewPashkin/python-tempo/

CHAPTER 2

# Features

- Flexible schedule model, that can express shcedules, that other libraries can't.
- Queries: containment of a single timestamp, future occurences.
- Bindings:
  - PostgreSQL
    - Domain type for storing schedules
    - Procedures for performing tests on them (timestamp containment, future occurences).
  - Django
    - Model field
    - Custom lookups (timestamp containment, intersection with interval between two timestamps, test if scheduled event occurs within given interval between two timestamps).
  - Django-REST-Framework
    - Serializer field for serializing and deserializing schedules.

CHAPTER 3

# Quick example

Just a short example, which shows, how to construct and query a schedule.

```
>>> import datetime as dt
>>> from itertools import islice
>>> from tempo.recurrenteventset import RecurrentEventSet
>>> recurrenteventset = RecurrentEventSet.from_json(
...     ('OR',
...         ('AND', [1, 5, 'day', 'week'], [10, 19, 'hour', 'day']),
...         ('AND', [5, 6, 'day', 'week'], [10, 16, 'hour', 'day']))
... )  # 10-19 from Monday to Thursday and 10-16 in Friday
>>> d1 = dt.datetime(year=2000, month=10, day=5, hour=18)
>>> d1.weekday()  # Thursday
3
>>> d1 in recurrenteventset
True
>>> d2 = dt.datetime(year=2000, month=10, day=6, hour=18)
>>> d2.weekday()  # Friday
4
>>> d2 in recurrenteventset
False
>>> d = dt.datetime(year=2000, month=1, day=1)
>>> list(islice(recurrenteventset.forward(start=d), 3))
[(datetime.datetime(2000, 1, 3, 10, 0),
  datetime.datetime(2000, 1, 3, 19, 0)),
 (datetime.datetime(2000, 1, 4, 10, 0),
  datetime.datetime(2000, 1, 4, 19, 0)),
 (datetime.datetime(2000, 1, 5, 10, 0),
  datetime.datetime(2000, 1, 5, 19, 0))]
```

CHAPTER 4

# Schedule model

## 4.1 Example

Here is an example of how Tempo represents schedules:

```
('OR',
        ('AND', [1, 5, 'day', 'week'], [10, 19, 'hour', 'day']),
        ('AND', [5, 6, 'day', 'week'], [10, 16, 'hour', 'day'])))
```

It means "from monday to thursday between 10am and 7pm and in friday between 10am and 4pm".

## 4.2 Informal definition

Basic building block of schedule is a recurrent event, which is defined is such way:

```
[<start time>, <end time>, <time unit>, <recurrence unit>]
```

*<start time>* and *<end time>* are numbers, that defines interval in which event takes it's place. *<time unit>* defines a unit of measurement of time for values of the interval. And *<recurrence unit>* defines how often the interval repeats. *<time unit>* and *<recurrence unit>* values are time measurement units, such as 'second', 'hour', 'day', 'week', 'year', etc. *<recurrence unit>* also can be 'null', which means, that the interval doesn't repeats in time, it just defines two points in time, that corresponds to start and end points of the event.

Recurrent events can be composed, using operators: union - *or*, intersection - *and* and negation - *not*.

CHAPTER 5

# Alternatives

- python-dateutil
- croniter

CHAPTER 6

# TODO

1. More tests for `RecurrentEventSet`.
2. Implement negative indexing for schedules - indexing from an end of a day or month, etc. It will make library able to model schedules like "last friday of the month".

CHAPTER 7

# Documentation

Contents:

## 7.1 Prerequisites

**Python** 2.7, 3, 3.2, 3.3, 3.4

**PostgreSQL** 9.4+

**Django** 1.7+

**REST Framework** 3+

## 7.2 Installation

### 7.2.1 Python

Just use PIP:

```
pip install python-tempo
```

### 7.2.2 PostgreSQL

1. Install PostgreSQL flavor of the library in the Python environment, that your PostgreSQL instance uses, so PL/Python stored procedures would be able to *import tempo*:

```
pip install python-tempo[postgresql]
```

2. Ensure, that PL/Python language is available for you in your PostgreSQL instance (see details here).
3. After installing Python egg, two commands will become available to you: `tempo-postgresql-install` and `tempo-postgresql-uninstall`. They are output to stdout installation and uninstallation SQL scripts respectively.

### 7.2.3 Django

Perform steps for PostgreSQL.

### 7.2.4 Django-REST-Framework

Perform steps for Python or PostgreSQL. Perform steps for PostgreSQL.

## 7.3 Usage

Look at *Schedule model* before reading this.

### 7.3.1 Python

Python API contains two classes: `RecurrentEvent` and `RecurrentEventSet`.

#### Simple schedules via *ReccurrentEvent* class

`RecurrentEvent` models simple schedules like "from 10 to 19 hours of the day", "from 1 to 4 months of the year", etc. For example this will define schedule "from first to sixth day of the week":

```
>>> from tempo.recurrentevent import RecurrentEvent
>>> recurrentevent = RecurrentEvent(1, 6, 'day', 'week')
```

Then, we can perform queries - containment of a single date:

```
>>> import datetime as dt
>>> d1 = dt.datetime(2000, 1, 10)
>>> d1.weekday()  # Monday - first day of the week
0
>>> d1 in recurrentevent
True
>>> d2 = dt.datetime(2000, 1, 14)
>>> d2.weekday()  # Friday - fifth day of the week
4
>>> d2 in recurrentevent
True
>>> d3 = dt.datetime(2000, 1, 15)
>>> d3.weekday()  # Saturday - sixth day of the week
5
>>> d3 in recurrentevent
False
```

We also can query for further occurences starting from certain point of time:

```
>>> from itertools import islice
>>> start = dt.datetime(2000, 1, 4)
>>> start.weekday()  # Tuesday
1
>>> list(islice(recurrentevent.forward(start), 3))
[(datetime.datetime(2000, 1, 4, 0, 0),
  datetime.datetime(2000, 1, 8, 0, 0)),
 (datetime.datetime(2000, 1, 10, 0, 0),
  datetime.datetime(2000, 1, 15, 0, 0)),
 (datetime.datetime(2000, 1, 17, 0, 0),
  datetime.datetime(2000, 1, 22, 0, 0))]
```

`RecurrentEvent.forward()` returns a generator, that yields as largest possible interval each time. In this case it's a time span between a monday and a saturday (non-inclusive) of each week.

Notice - *start* defines Tuesday, but our schedule starts on Monday - and `forward()`, yielded the first time interval, that starts on Tuesday, the time, that equals our *start* argument. It shrinked the first time interval by the *start*, since otherwise the first time interval would be started from the time earlier, than *start*. We can change this behaviour, by passing additional argument *trim*:

```
>>> list(islice(recurrentevent.forward(start, trim=False), 3))
[(datetime.datetime(2000, 1, 3, 0, 0),
  datetime.datetime(2000, 1, 8, 0, 0)),
 (datetime.datetime(2000, 1, 10, 0, 0),
  datetime.datetime(2000, 1, 15, 0, 0)),
 (datetime.datetime(2000, 1, 17, 0, 0),
  datetime.datetime(2000, 1, 22, 0, 0))]
```

Now `RecurrentEvent.forward()` yielded largest possible interval not only in future direction for the *start*, but also in past direction.

#### Composite schedules via *ReccurrentEvent* class

Let's now take a look at `RecurrentEventSet`. It makes possible to compose simple schedules to more complex ones, using operators of union (*OR*), intersection (*AND*) and negation (*NOT*).

For example:

```
>>> from tempo.recurrenteventset import RecurrentEventSet
>>> recurrenteventset = RecurrentEventSet.from_json(
...     ('OR',
...         ('AND',
...             ('NOT', [12, 13, 'hour', 'day']),
...             ('AND', [1, 4, 'day', 'week'], [10, 19, 'hour', 'day']),
...             ('AND', [5, 6, 'day', 'week'], [10, 16, 'hour', 'day']))))
... )
```

That defines "from Monday to Thursday from 10am to 7pm and in Friday from 10am to 4pm with the gap from 12am to 1pm".

`RecurrentEventSet` has the same interface as `RecurrentEvent`: it provides `RecurrentEventSet.forward()` and `RecurrentEventSet.__contains__()` methods, which has exactly the same meaning as `RecurrentEvent` ones has.

**Note:** Here, documentation uses `RecurrentEventSet.from_json()`, alternative constructor, it's because of convenience. `RecurrentEventSet` has also a regular constructor, which expects an *expression* of the same structure, but with `RecurrentEvent` instances instead of their JSON representations.

### 7.3.2 PostgreSQL

The library provides domain types and functions, that represents lirary's classes and their methods, which has similar to Python's methods signatures.

**Note:** They are actually bindings to Python library, not imlementations from scratch, that's why user required to have Python library installed and available for import from PL/Python procedures.

Currently only methods for `RecurrentEventSet` are supported.

### 7.3.3 Django

`fields.RecurrentEventSetField` is a Django model field. It has adds no additional parameters, to the standard ones. It supports a number of custom lookups:

1. *contains* - tests a single `datetime.datetime` object for containment.
2. *intersects* - tests a pair of `datetime.datetime` objects for intersection with a time defined by a schedule.
3. *occurs_within* - tests some of time intervals, defined by a schedule, included in a boundaries, defined by a pair of `datetime.datetime` objects.

Let's take movies as an example, and that's a Django model, that describes a movie:

```
from django.db import models
from tempo.django.fields import RecurrentEventSetField


class Movie(models.Model):
    name = models.CharField('Name', max_length=99)
    schedule = RecurrentEventSetField('Schedule')

    __str__ = __unicode__ = lambda self: self.name
```

Then, populate the database:

```
>>> Movie.objects.create(name='Titanic',
...                      schedule=['OR', [11, 14, 'hour', 'day']])
<Movie: Titanic>
>>> Movie.objects.create(name='Lord of the Rings',
...                      schedule=['OR', [12, 15, 'hour', 'day']])
<Movie: Lord of the Rings>
>>> Movie.objects.create(name='Avatar',
...                      schedule=['OR', [18, 20, 'hour', 'day']])
<Movie: Avatar>
```

With *contains* lookup, we can check, what movies a running in a certain point of time, for example - in *2015-01-01 13:00*:

```
>>> import datetime as dt
>>> d = dt.datetime(2015, 1, 1, 13)
>>> Movie.objects.filter(schedule__contains=d).order_by('name')
[<Movie: Lord of the Rings>, <Movie: Titanic>]
```

With *intersects* lookup, we can find, what movies will be running in given time period, for example - from *2015-01-01 14:00* to *2015-01-01 20:00*:

```
>>> interval = (dt.datetime(2015, 1, 1, 14), dt.datetime(2015, 1, 1, 20))
>>> Movie.objects.filter(schedule__intersects=interval).order_by('name')
[<Movie: Avatar>, <Movie: Lord of the Rings>]
```

And with *occurs_within* lookup, we can find, what movies we can watch from a start to an end in certain period of time, for example - from *2015-01-01 10:00* to *2015-01-01 19:00*:

```
>>> interval = (dt.datetime(2015, 1, 1, 10), dt.datetime(2015, 1, 1, 19))
>>> Movie.objects.filter(schedule__occurs_within=interval).order_by('name')
[<Movie: Lord of the Rings>, <Movie: Titanic>]
```

### 7.3.4 Django-REST-Framework

Django REST Framework binding provides a custom serializer field - `serializers.RecurrentEventSetField`. It's very simple and adds no additional parameters. Just refer to DRF serializers documentation and use this field like any other serialzier field.

## 7.4 API reference

### 7.4.1 Python

#### tempo.recurrentevent

Provides RecurrentEvent class.

**class** `tempo.recurrentevent.`**`RecurrentEvent`**(*start*, *stop*, *unit*, *recurrence=None*)

An interval of time expressed in some 'unit' of time (second, week, year, etc), recurring with some 'recurrence', also expressed in some unit of time. For example minutes interval can recur hourly or yearly, but can't recur secondly.

With *None* passed as 'recurrence', time interval will be defined without recurrence, just as a single non-recurring interval between two points in time and counted from "the beginning of time". By convention "the beginning of time" is 1-1-1 00:00:00.

**Parameters**

- **`start`** (*int*) – Start of recurring interval.
- **`stop`** (*int*) – Non-inclusive end of recurring interval.
- **`unit`** (*str*) – Unit of time in which time interval is expressed.
- **`recurrence`** (*str, optional*) – Recurrence of time interval. Can be (and by default is) *None*, which means - "no recurrence".

**Examples**

```
>>> from datetime import datetime
>>> recurrentevent = RecurrentEvent(0, 15, Unit.SECOND, Unit.MINUTE)
>>> datetime(2000, 1, 1, 5, 3, 10) in recurrentevent
... True
>>> datetime(2000, 1, 1, 5, 3, 16) in recurrentevent
... False
```

**`__contains__`**(*item*)

Test given datetime 'item' for containment in the recurrent event.

**Parameters** **`item`** (*datetime.datetime*) – A 'datetime' object to test.

**Returns** Result of containment test.

**Return type** bool

**Notes**

The algorithm here consists of following steps:

If recurrence is set:

1. Given datetime floored to unit of 'recurrence' and stored.
2. Then given datetime floored to unit of 'unit' and stored.
3. Delta between resulting datetime objects is calculated and expressed in units of 'unit'. For example if delta is "2 days" and 'unit' is minutes, delta will be "2*24*60 minutes".

If recurrence is not set:

1. Delta between date of "the beginning of time" and given date is calculated and expressed in units of 'unit'.
4. Resulting delta tested for containment in the interval.

**forward**(*start*, *trim=True*)

Iterate time intervals starting from 'start'. Intervals returned in form of *(start, end)* pair, where *start* is a datetime object representing the start of the interval and *end* is the non-inclusive end of the interval.

**Parameters**

- **start** (*datetime.datetime*) – A lower bound for the resulting sequence of intervals.
- **trim** (*bool*) – Whether a first interval should be trimmed by 'start' or it should be full, so it's start point may potentially be earlier, that 'start'.

**Yields**

- **start** (*datetime.datetime*) – Start of an interval.
- **end** (*datetime.datetime*) – End of an interval.

**classmethod from_json**(*value*)

Constructs *RecurrentEvent* instance from JSON serializable representation or from JSON string.

**isgapless**()

Tests if the RecurrentEvent instance defines infinite time interval.

**to_json**()

Exports *RecurrentEvent* instance to JSON serializable representation.

### tempo.recurrenteventset

Provides RecurrentEventSet class.

**class** tempo.recurrenteventset.**RecurrentEventSet**(*expression*)

A set of time intervals, combined with a set logic operators: AND, OR and NOT.

**Parameters** **expression** (*tuple*) – A nested expression, composed of operators and arguments, which are *RecurrentEvent* instances or sub-expressions. Example of an expression:

```
(AND,
    RecurrentEvent(Interval(10, 19), 'hour', 'day'),
    (NOT, RecurrentEvent(Interval(14, 15), 'hour', 'day')),
    (NOT, RecurrentEvent(Interval(6, 7), 'day', 'week')),
])
```

It means: 'From 10:00 to 19:00 every day, except from 14:00 to 15:00, and weekends'.

**__contains__**(*item*)

Containment test. Accepts whatever RecurrentEvent can test for containment.

**forward**(*start*, *trim=True*)

Generates intervals according to the expression.

Intervals never overlap. Each next interval is largest possbile interval.

**Parameters**

- **start** (*datetime.datetime*) – Inclusive start date.
- **trim** (*bool*) – If *True* (which is default), the starting point of a first interval will always be equal to or greater than'start'. Otherwise it will be equal to the point, where the interval actually starts, which may be placed earlier in time, that 'start'.

**Yields** *tuple* – Inclusive start and non-inclusive dates of an interval.

#### Notes

The alghorithm is simple:

1. It generates intervals from RecurrentEvent instances and applies set logic operators on them.
2. Checks if resulting interval has gap.
3. Checks if there is a possibility, that this gap will gone, by checking if some of the generators could possibly generate interval that will intersect with gap.
4. If checks succeed, yields interval previous to gap.
5. If not - iterates generators until check succeed.

This implementation if fairly ineffective and should be otimized.

**classmethod from_json**(*value*)

Constructs *RecurrentEventSet* instance from JSON serializable representation or from JSON string.

**static from_json_callback**(*operator*, **args*)

Converts arguments that are time intervals to Python.

**to_json**()

Exports *RecurrentEventSet* instance to JSON serializable representation.

**static to_json_callback**(*operator*, **args*)

Converts arguments that are time intervals to JSON.

**static validate_json**(*expression*)

Validates JSON expression.

**class** tempo.recurrenteventset.**Result**(*value*)

Callback result wrapper.

Intended to be used to avoid confusion between expressions and callback results, in case if they are expressions themselves.

### 7.4.2 PostgreSQL

**tempo_recurrentevent**

**TYPE** domain type

**BASE** jsonb

A domain type, that represents `RecurrentEvent`.

**tempo_recurrenteventset**

**TYPE** domain type

**BASE** jsonb

A domain type, that represents `RecurrentEventSet`.

**tempo_recurrenteventset_contains (recurrenteventset tempo_recurrenteventset, datetime times**

**TYPE** function

**RETURNS** boolean

**VOLATILITY** IMMUTABLE

**LANGUAGE** plpythonu

Checks *datetime* for containment in *recurrenteventset*.

**tempo_recurrenteventset_forward (recurrenteventset tempo_recurrenteventset, start timestamp**

**TYPE** function

**RETURNS** TABLE(start timestamp, stop timestamp)

**VOLATILITY** IMMUTABLE

**LANGUAGE** plpythonu

Future intervals of *recurrenteventset* as set of rows.

### 7.4.3 Django

#### tempo.django.fields

Provides Django model fields API for RecurrentEventSet.

**class** `tempo.django.fields.`**`Contains`**(*lhs*, *rhs*)

Provides *contains* lookup for `RecurrentEventSetField`.

Checks a single `datetime` object for containment in `RecurrentEventSet`.

**class** `tempo.django.fields.`**`Intersects`**(*lhs*, *rhs*)

Provides *intersects* lookup for `RecurrentEventSetField`.

Checks a given time interval in form of a pair-tuple of `datetime` objects, intersects with time defined by time interval set in given column.

**class** `tempo.django.fields.`**`OccursWithin`**(*lhs*, *rhs*)

Provides *occurs_within* lookup for `RecurrentEventSetField`.

Checks if some of continous events, defined in time interval set is enclosed by dates in given pair-tuple of datetime objects.

**class** `tempo.django.fields.`**`RecurrentEventSetField`**(*verbose_name=None, name=None, primary_key=False, max_length=None, unique=False, blank=False, null=False, db_index=False, rel=None, default=<class django.db.models.fields.NOT_PROVIDED>, editable=True, serialize=True, unique_for_date=None, unique_for_month=None, unique_for_year=None, choices=None, help_text=u'', db_column=None, db_tablespace=None, auto_created=False, validators=[], error_messages=None*)

DB representation of recurrenteventset. Requires PostgreSQL 9.4+.

### 7.4.4 Django-REST-Framework

#### tempo.rest_framework.serializers

Provides utilities for serialization/deserialization of Tempo data types.

**class** `tempo.rest_framework.serializers.`**`RecurrentEventSetField`**(*read_only=False, write_only=False, required=None, default=<class rest_framework.fields.empty>, initial=<class rest_framework.fields.empty>, source=None, label=None, help_text=None, style=None, error_messages=None, validators=None, allow_null=False*)

Representation of RecurrentEventSet.

CHAPTER 8

# Indices and tables

- genindex
- modindex
- search

# Python Module Index

## t

tempo.django.fields, 22
tempo.recurrentevent, 19
tempo.recurrenteventset, 20
tempo.rest_framework.serializers, 23

## Symbols

__contains__() (tempo.recurrentevent.RecurrentEvent method), 19
__contains__() (tempo.recurrenteventset.RecurrentEventSet method), 20

## C

Contains (class in tempo.django.fields), 22

## F

forward() (tempo.recurrentevent.RecurrentEvent method), 20
forward() (tempo.recurrenteventset.RecurrentEventSet method), 20
from_json() (tempo.recurrentevent.RecurrentEvent class method), 20
from_json() (tempo.recurrenteventset.RecurrentEventSet class method), 21
from_json_callback() (tempo.recurrenteventset.RecurrentEventSet static method), 21

## I

Intersects (class in tempo.django.fields), 22
isgapless() (tempo.recurrentevent.RecurrentEvent method), 20

## O

OccursWithin (class in tempo.django.fields), 22

## R

RecurrentEvent (class in tempo.recurrentevent), 19
RecurrentEventSet (class in tempo.recurrenteventset), 20
RecurrentEventSetField (class in tempo.django.fields), 22
RecurrentEventSetField (class in tempo.rest_framework.serializers), 23
Result (class in tempo.recurrenteventset), 21

## T

tempo.django.fields (module), 22
tempo.recurrentevent (module), 19
tempo.recurrenteventset (module), 20
tempo.rest_framework.serializers (module), 23
to_json() (tempo.recurrentevent.RecurrentEvent method), 20
to_json() (tempo.recurrenteventset.RecurrentEventSet method), 21
to_json_callback() (tempo.recurrenteventset.RecurrentEventSet static method), 21

## V

validate_json() (tempo.recurrenteventset.RecurrentEventSet static method), 21