timeboard Documentation

Release 0.2

Maxim Mamaev

Jun 25, 2022

Contents:

`timeboard` performs calendar calculations over business schedules such as business days or work shifts.

CHAPTER 1

About timeboard

`timeboard` creates schedules of work periods and performs calendar calculations over them. You can build standard business day calendars as well as a variety of other schedules, simple or complex.

Examples of problems solved by `timeboard`:

- If we have 20 business days to complete the project, when will be the deadline?
- If a person was employed from November 15 to December 22 and salary is paid monthly, how many month's salaries has the employee earned?
- The above-mentioned person was scheduled to work Mondays, Tuesdays, Saturdays, and Sundays on odd weeks, and Wednesdays, Thursdays, and Fridays on even weeks. The question is the same.
- A 24x7 call center operates in shifts of varying length starting at 02:00, 08:00, and 18:00. An operator comes in on every fourth shift and is paid per shift. How many shifts has the operator sat in a specific month?
- With employees entering and leaving a company throughout a year, what was the average annual headcount?

Based on pandas timeseries library, `timeboard` gives more flexibility than pandas's built-in business calendars. The key features of `timeboard` are:

- You can choose any time frequencies (days, hours, multiple-hour shifts, etc.) as work periods.
- You can create sophisticated schedules which can combine periodical patterns, seasonal variations, stop-and-resume behavior, etc.
- There are built-in standard business day calendars (in this version: for USA, UK, and Russia).

1.1 Contributing

`timeboard` is authored and maintained by Maxim Mamaev.

Please use Github issues for the feedback.

1.2 License

```
3-Clause BSD License

Copyright (c) 2018, Maxim Mamaev
All rights reserved.

Redistribution and use in source and binary forms, with or without
modification, are permitted provided that the following conditions are met:

1. Redistributions of source code must retain the above copyright notice,
this list of conditions and the following disclaimer.

2. Redistributions in binary form must reproduce the above copyright
notice, this list of conditions and the following disclaimer in the
documentation and/or other materials provided with the distribution.

3. Neither the name of the copyright holder nor the names of its
contributors may be used to endorse or promote products derived from this
software without specific prior written permission.

THIS SOFTWARE IS PROVIDED BY THE COPYRIGHT HOLDERS AND CONTRIBUTORS "AS
IS" AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO,
THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR
PURPOSE ARE DISCLAIMED. IN NO EVENT SHALL THE COPYRIGHT HOLDER OR
CONTRIBUTORS BE LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL,
EXEMPLARY, OR CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT LIMITED TO,
PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES; LOSS OF USE, DATA, OR
PROFITS; OR BUSINESS INTERRUPTION) HOWEVER CAUSED AND ON ANY THEORY OF
LIABILITY, WHETHER IN CONTRACT, STRICT LIABILITY, OR TORT (INCLUDING
NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT OF THE USE OF THIS
SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.
```

1.3 Attribution

Logo design by Olga Mamaeva.

Icon ‘Worker’ made by Freepik from www.flaticon.com is used as an element of the logo.

CHAPTER 2

Installation

2.1 Python version support

`timeboard` is tested with Python versions 2.7, 3.6, 3.7, and 3.8.

2.2 Installation

```
pip install timeboard
```

The import statement to run all the examples is:

```
>>> import timeboard as tb
```

2.3 Dependencies

Package	versions tested
pandas	0.22 - 1.0
numpy	1.13 - 1.18
python-dateutil	2.6.1 - 2.8.1
six	1.11 - 1.14

The code is tested by pytest .

CHAPTER 3

Quick Start Guide

3.1 Set up a timeboard

To get started you need to build a timeboard (calendar). The simplest way to do so is to use a preconfigured calendar which is shipped with the package. Let's take a regular business day calendar for the United States.

```
>>> import timeboard.calendars.US as US
>>> clnd = US.Weekly8x5()
```

Note: If you need to build a custom calendar, for example, a schedule of shifts for a 24x7 call center, *Making a Timeboard* section of the documentation explains this topic in details.

Once you have got a timeboard, you may perform queries and calculations over it.

3.2 Play with workshifts

Calling a timeboard instance *clnd* with a single point in time produces an object representing a unit of the calendar (in this case, a day) that contains this point in time. Object of this type is called *workshift*.

Is a certain date a business day?

```
>>> ws = clnd('27 May 2017')
>>> ws.is_on_duty()
False
```

Indeed, it was a Saturday.

When was the next business day?

```
>>> ws.rollforward()
Workshift(6359) of 'D' at 2017-05-30
```

The returned calendar unit (workshift) has the sequence number of 6359 and represents the day of 30 May 2017, which, by the way, was the Tuesday after the Memorial Day holiday.

If we were to finish the project in 22 business days starting on 01 May 2017, when would be our deadline?

```
>>> clnd('01 May 2017') + 22
Workshift(6361) of 'D' at 2017-06-01
```

This is the same as:

```
>>> clnd('01 May 2017').rollforward(22)
Workshift(6361) of 'D' at 2017-06-01
```

3.3 Play with intervals

Calling `clnd()` with a different set of parameters produces an object representing an *interval* on the calendar. The interval below contains all workshifts of the months of May 2017.

How many business days were there in a certain month?

```
>>> may2017 = clnd('May 2017', period='M')
>>> may2017.count()
22
```

How many days off?

```
>>> may2017.count(duty='off')
9
```

How many working hours?

```
>>> may2017.worktime()
176.0
```

An employee was on the staff from April 3, 2017 to May 15, 2017. **What portion of April's salary did the company owe them?**

Calling `clnd()` with a tuple of two points in time produces an interval containing all workshifts between these points, inclusively.

```
>>> time_in_company = clnd(('03 Apr 2017','15 May 2017'))
>>> time_in_company.what_portion_of(clnd('Apr 2017', period='M'))
1.0
```

Indeed, the 1st and the 2nd of April in 2017 fell on the weekend, therefore, having started on the 3rd, the employee checked out all the working days in the month.

And what portion of May's?

```
>>> time_in_company.what_portion_of(may2017)
0.5
```

How many days had the employee worked in May?

The multiplication operator returns the intersection of two intervals.

```
>>> (time_in_company * may2017).count()
11
```

How many hours?

```
>>> (time_in_company * may2017).worktime()
88
```

An employee was on the staff from 01 Jan 2016 to 15 Jul 2017. **How many years this person had worked for the company?**

```
>>> clnd(('01 Jan 2016', '15 Jul 2017')).count_periods('A')
1.5421686746987953
```

CHAPTER 4

Data Model

Table of Contents

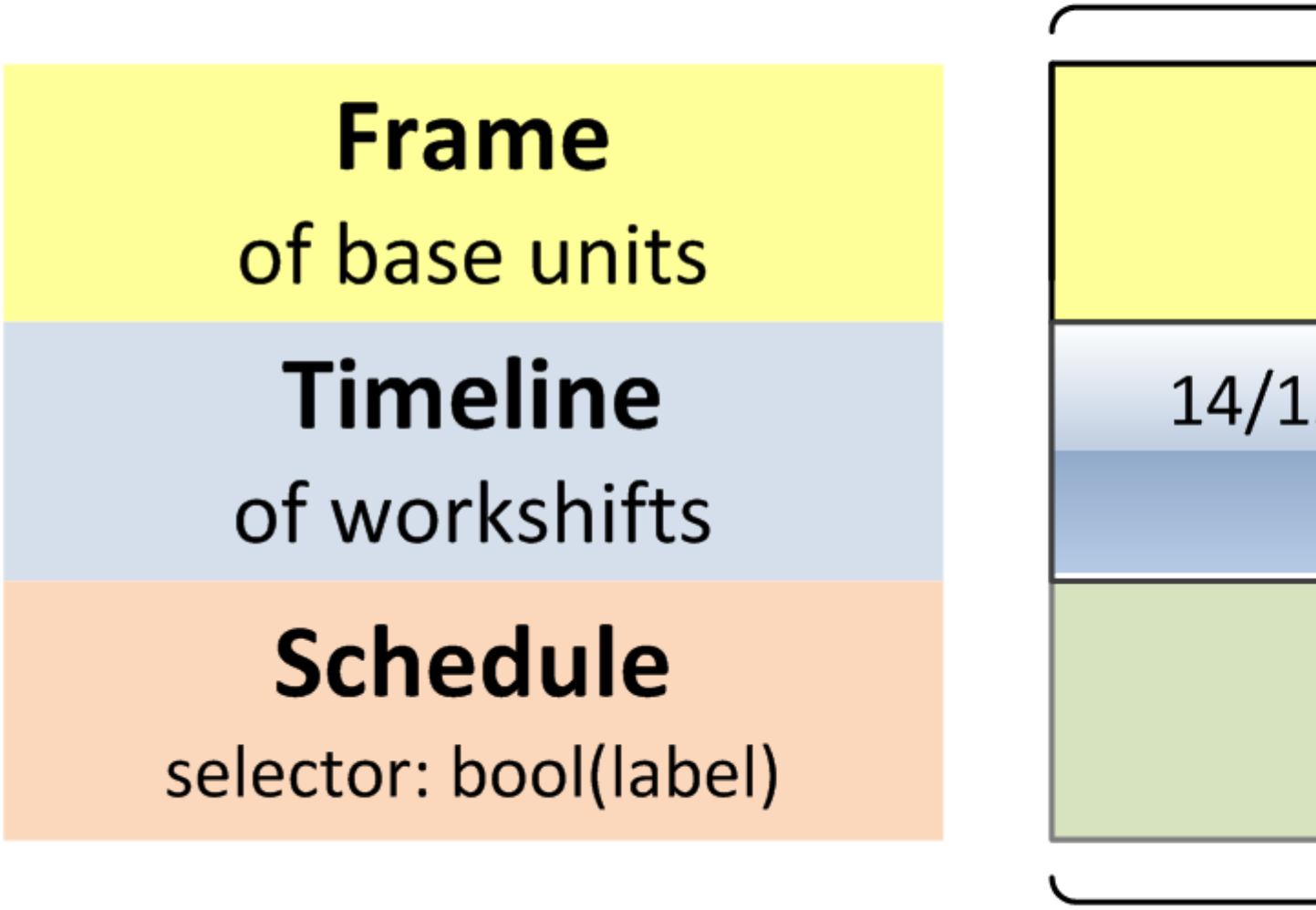
4.1 Timeboard

Timeboard is a representation of a custom business calendar.

More precisely, **timeboard** is a collection of work *schedules* based on a specific *timeline* of *workshifts* built upon a reference *frame*.

Note: The terms *workshift*, *frame*, *timeline*, and *schedule* have exact meanings that are explained below. On the other hand, word *calendar* is considered rather ambiguous. It is used occasionally as a loose synonym for *timeboard* when there is no risk of misunderstanding.

Timeboard is the upper-level object of the data model. You use timeboard as the entry point for all calculations.


Frame
of base units
Timeline
of workshifts
Schedule
selector: bool(label)
14/1

4.2 Workshift

Workshift is a period of time during which a business agent is either active or not.

No assumptions are made about what "business" is and who its "agents" may be. It could be regular office workers, or operators in a 24x7 call center, or trains calling at a station on specific days.

The activity state of a workshift is called **duty**; therefore for a given business agent the workshift is either "on duty" or "off duty". It is not relevant for determining the duty whether the agent is continuously active through the workshift, or takes breaks, or works only for a part of the workshift. The duty is assigned to a workshift as a whole.

It is up to the user to define and interpret periods of time as workshifts in a way which is suitable for user's application. For example, when making plans on the regular calendar of 8-hour business days, you do not care about exact working hours. Hence, it is sufficient to designate whole days as workshifts with business days being on-duty workshifts, and weekends and holidays being off-duty. On the other hand, to build working schedules for operators in a 24x7 call center who work in 8-hour shifts you have to designate each 8-hour period as a separate workshift.

See also *Work time* section below which discusses counting the actual work time.

Note that both on-duty and off-duty periods are called "workshifts" although *work*, whatever it might be, is not carried out by the business agent when the duty is off. Generally speaking, though not necessarily, some other business agent may operate a reversed schedule, doing *work* when the first agent is idle. For example, this is the case for a 24x7 call center.

4.3 Frame and Base Units

The span of time covered by the timeboard is represented as a reference frame. **Frame** is a monotonous sequence of uniform periods of time called **base units**. The base unit is atomic, meaning that everything on the timeboard consists of an integer number of base units.

4.4 Timeline

Timeline is a continuous sequence of workshifts laid upon the frame.

Each workshift consists of one or more base units of the frame. The number of base units constituting a workshift is called the **duration** of the workshift. Different workshifts do not necessarily have the same duration.

Each base unit of the frame belongs to one and only one workshift. This means that workshifts do not overlap and there are no gaps between workshifts.

Each workshift is given a **label**. The type and semantic of labels are application-specific. Several or all workshifts can be assigned the same label.

Labels are essential for defining the duty of the workshifts.

4.5 Interval

Interval is a continuous series of workshifts within the timeline. Number if workshifts in an interval is called the **length** of the interval. Interval contains at least one workshift.

4.6 Schedule

Schedule defines duty status of each workshift on the timeline according to a rule called **selector**. Selector examines the label of a workshift and returns True if this workshift is considered on duty under this schedule, otherwise, it returns False.

Timeboard contains at least one ("default") schedule and may contain many. Each schedule employs its specific selector. The default selector for the default schedule returns `bool(label)`.

The duty of a workshift may vary depending on the schedule.

For example, let the timeline consist of calendar days where weekdays from Monday through Friday are labeled with 2, Saturdays are labeled with 1, and Sundays and holidays are labeled with 0. The schedule of regular business days is obtained by applying selector `label>1`, and under this schedule, Saturdays are off duty. However, if children attend schools on Saturdays, the school schedule can be obtained from the same timeline with selector *label>0*. Under the latter schedule, Saturdays are on duty.

4.7 Compound Workshifts

It is important to emphasize that, when working with a timeboard, you reason about workshifts rather than base units. Duty is associated with a workshift, not with a base unit. All calendar calculations are performed either on workshifts or on intervals.

In many cases, workshift coincides with a base unit. The only reason to be otherwise is when you need workshifts of varying duration. Let's illustrate this point with examples.

When reasoning about business days, the time of day when working hours start or end is not relevant. For the purpose of the calendar, it suffices to label a whole weekday on-duty in spite of the fact that only 8 hours of 24 are actually taken by business activity.

Moreover, if the number of working hours do vary from day to day (i.e. Friday's hours are shorter) and you need to track that, the task can be solved just with workshift labeling. A workshift still takes a whole day, and within week workshifts are labeled `[8, 8, 8, 8, 7, 0, 0]` reflecting the number of working hours. The default `selector=bool(label)` works fine with that. Therefore, while actual workshifts do have varying duration, you do not *need* to model this in the timeline. You can use a simpler timeboard where each workshift correspond to a base unit of one calendar day.

Now consider the case of a 24x7 call center operating in 8-hour shifts. Clearly, a workshift is to be represented by an 8-hour period but this does not necessarily call for workshifts consisting of 8 base units, each base unit one hour long. When building the frame, you are not limited to use of base units equal to a single calendar period, i.e. one hour, one day, and so on. You can take a base unit which spans multiple consecutive calendar periods, for example, 8 hours. Therefore, in this case, there is still no need to create workshifts consisting of several base units, as 8-hour base units can be directly mapped to 8-hour workshifts.

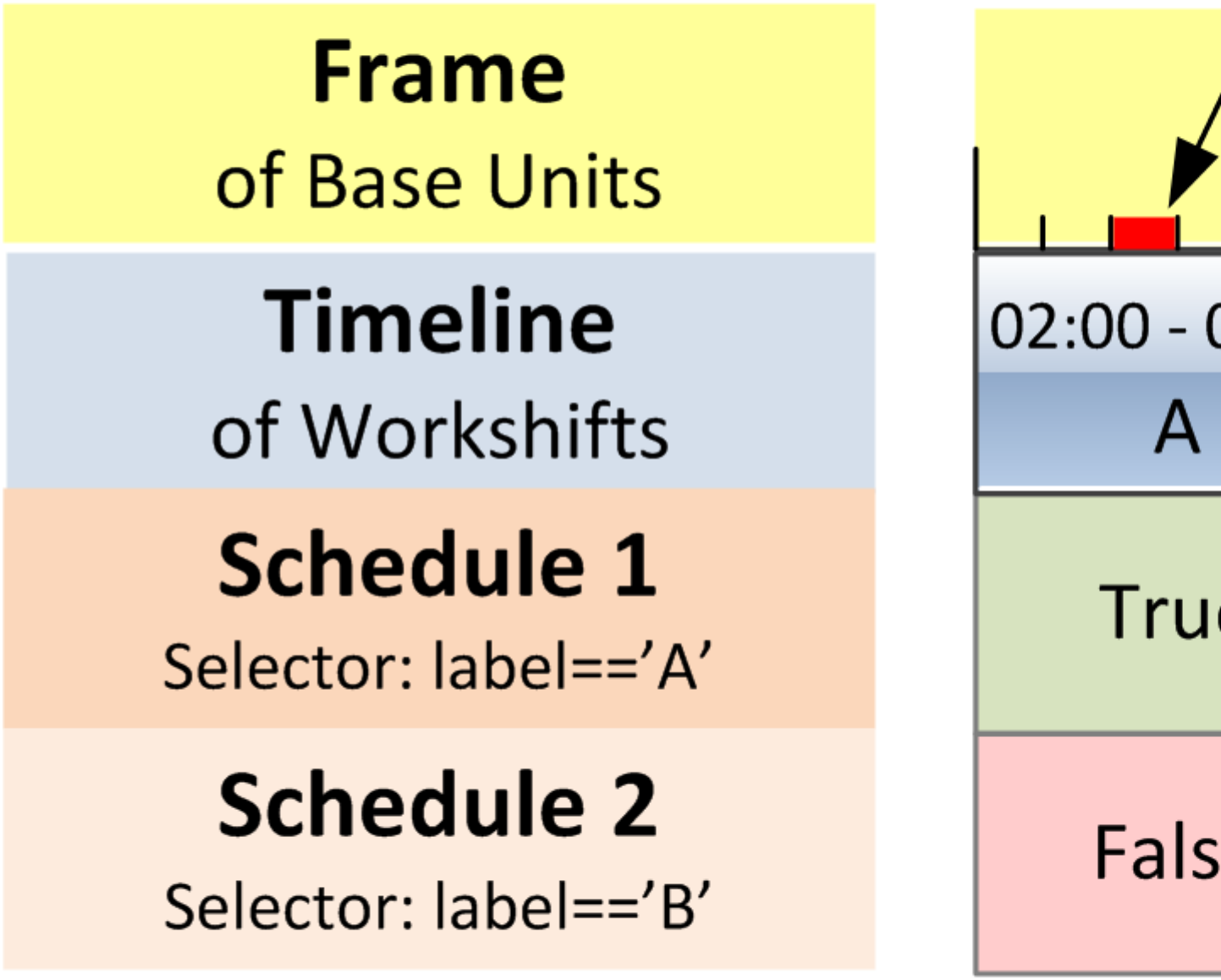
However, the things change if we assume that the call center operates shifts of varying durations, i.e. 08:00 to 18:00 (10 hours), 18:00 to 02:00 (8 hours), and 02:00 to 08:00 (6 hours).

Now the base unit has to be a common divisor of all workshift durations which is one hour. (Technically, it also can be two hours, which does not make the case any simpler, so we will stick to the more natural one-hour clocking.)

This case cannot be elegantly handled by workshifts bound to base units. This way we would end up, for any day, not with three workshifts of 10, 8 and 6 hours long but with a succession of 24 one-hour workshifts of which either 10, 8 or 6 consecutive ones will be labeled on-duty. Creating meaningful work schedules and performing calculations for such timeline would be a rather cumbersome challenge. Therefore we have to decouple workshifts from base units and create the timeline where individual workshifts have durations of 10, 8, and 6 base units in the repeating pattern.

Having said that, while in many cases a workshift will coincide with a base unit, these entities have different purposes.

A workshift comprising more than one base unit is called **compound workshift**.


Frame
of Base Units
Timeline
of Workshifts
Schedule 1
Selector: label==’A’
Schedule 2
Selector: label==’B’
02:00 -
A
Tru
Fals

4.8 Work time

Work time (also spelled 'worktime' in names of functions and parameters) is the amount of time within workshift which the agent spends actually doing work. In many use cases, you will want to find out the work time of a specific workshift or the total work time of an interval.

Depending on the model of a timeboard, the duration of workshift may or may not represent the work time. Typically, in the models based on continuous succession of shifts, the work time takes the entire workshift. On the other hand, in calendars of business days, the actual work time takes only a part of a workshift (that is, of a day).

In the latter case, you may use workshift labels to indicate the work time as it has been shown in the previous section. Obviously, such labels must be numbers. Their interpretation is up to the user.

When creating your timeboard you will have to specify the source of information for counting the work time: either it is workshift's duration or workshift's label. Accordingly, the functions counting the work time will return either the number of base units in the workshift/interval or the sum of the labels.

CHAPTER 5

Making a Timeboard

Table of Contents

- *Not all* Marker *frequencies are valid*
- *Alignment of frame may be critical*
- *Specific days of month*

A timeboard is constructed by calling `Timeboard()` constructor with parameters that define the desired configuration of the calendar. In the simplest case this can be done by a one-liner but most likely you will use auxiliary tools such as `Organizer`, `Marker`, and `RememberingPattern`.

The import statement to run the examples:

```
>>> import timeboard as tb
```

It is assumed that you are familiar with *Data Model*.

5.1 Basic case

`Timeboard` class requires four mandatory parameters for instantiating a timeboard:

```
>>> clnd = tb.Timeboard(base_unit_freq='D',
...                     start='01 Oct 2017', end='10 Oct 2017',
...                     layout=[1, 0, 0])
```

The first three parameters define the frame:

base_unit_freq [str] A pandas-compatible calendar frequency (i.e. 'D' for calendar day or '8H' for 8 consecutive hours regarded as one period) which defines timeboard's base unit. Pandas-native business periods (i.e. 'BM') are not supported.

start [*Timestamp*-like] A point in time referring to the first base unit of the timeboard. The point in time can be located anywhere within this base unit. The value may be a pandas Timestamp, or a string convertible to Timestamp (i.e. "01 Oct 2017 18:00"), or a datetime object.

end [*Timestamp*-like] Same as *start* but for the last base unit of the timeboard.

The fourth parameter, *layout*, describes the timeline of workshifts.

In the basic case *layout* is simply an iterable of workshift labels. In the above example `layout=[1, 0, 0]` means that each workshift occupies one base unit; the workshift at the first base unit receives label 1, the second workshift receives label 0, the third - again label 0. Further on, label assignment repeats in cycles: the forth workshift will get label 1, the fifth - 0, the sixth - 0, the seventh - 1, and so on. This way the timeline is created.

Under the hood, the timeboard builds default schedule using default selector which returns `bool(label)`. Therefore, under this schedule, the first and then every forth workshift are on duty, and the rest are off duty.

```
>>> print(clnd)
Timeboard of 'D': 2017-10-01 -> 2017-10-10

        ws_ref      start  duration        end  label  on_duty
loc
0   2017-10-01 2017-10-01         1 2017-10-01    1.0     True
1   2017-10-02 2017-10-02         1 2017-10-02    0.0    False
2   2017-10-03 2017-10-03         1 2017-10-03    0.0    False
3   2017-10-04 2017-10-04         1 2017-10-04    1.0     True
4   2017-10-05 2017-10-05         1 2017-10-05    0.0    False
5   2017-10-06 2017-10-06         1 2017-10-06    0.0    False
```

(continues on next page)

(continued from previous page)

```
6   2017-10-07 2017-10-07         1 2017-10-07    1.0     True
7   2017-10-08 2017-10-08         1 2017-10-08    0.0    False
8   2017-10-09 2017-10-09         1 2017-10-09    0.0    False
9   2017-10-10 2017-10-10         1 2017-10-10    1.0     True
```

5.1.1 Amendments

You use the optional parameter *amendments* to account for any disruptions of the regular pattern of the calendar (such as holidays, etc.).

amendments are a dictionary. The keys are *Timestamp*-like points in time used to identify workshifts (the point in time may be located anywhere within the workshift, i.e. at noon of a day as in the example below). The values of *amendments* are labels for the corresponding workshifts overriding the labels which have been set by *layout*.

```
>>> clnd = tb.Timeboard(base_unit_freq='D',
...                     start='01 Oct 2017', end='10 Oct 2017',
...                     layout=[1, 0, 0],
...                     amendments={'07 Oct 2017 12:00': 0})
>>> print(clnd)
Timeboard of 'D': 2017-10-01 -> 2017-10-10

        ws_ref      start  duration        end  label  on_duty
loc
0   2017-10-01 2017-10-01         1 2017-10-01      1     True
1   2017-10-02 2017-10-02         1 2017-10-02      0    False
2   2017-10-03 2017-10-03         1 2017-10-03      0    False
3   2017-10-04 2017-10-04         1 2017-10-04      1     True
4   2017-10-05 2017-10-05         1 2017-10-05      0    False
5   2017-10-06 2017-10-06         1 2017-10-06      0    False
6   2017-10-07 2017-10-07         1 2017-10-07      0    False
7   2017-10-08 2017-10-08         1 2017-10-08      0    False
8   2017-10-09 2017-10-09         1 2017-10-09      0    False
9   2017-10-10 2017-10-10         1 2017-10-10      1     True
```

Note, that if there are several keys in *amendments* which refer to the same workshift, the final label of this workshift would be unpredictable, therefore a *KeyError* is raised:

```
>>> clnd = tb.Timeboard(base_unit_freq='D',
...                     start='01 Oct 2017', end='10 Oct 2017',
...                     layout=[1, 0, 0],
...                     amendments={'07 Oct 2017 12:00': 0,
...                                 '07 Oct 2017 15:00': 1})
---------------------------------------------------------------------------
KeyError                                  Traceback (most recent call last)
  ...
KeyError: "Amendments key '07 Oct 2017 15:00' is a duplicate reference to workshift 6"
```

5.1.2 Other *Timeboard* parameters

workshift_ref [{`"start"` | `"end"`}, optional (default `"start"`)] Define what point in time will be used to represent a workshift. The respective point in time will be returned by `Workshift.to_timestamp()`. Available options: `"start"` to use the start time of the workshift, `"end"` to use the end time.

When printing a timeboard, the workshift reference time is shown in "ws_ref" column.

Workshift reference time is used to determine to which calendar period the workshift belongs if the workshift straddles a boundary of the calendar period. This is used by `Interval.count_periods()`.

default_name [str, optional] The name for the default schedule. If not supplied, "on_duty" is used.

When printing a timeboard, the rightmost column(s) are titled with the names of the schedules and show the workshift duty statuses under the corresponding schedules: True if the workshift is on duty, False otherwise. There is at least one column, showing the default schedule.

default_selector [function, optional] The selector function for the default schedule. This is the function which takes one argument - label of a workshift and returns True if this is an on-duty workshift, False otherwise. If not supplied, the function that returns `bool(label)` is used.

worktime_source [{`'duration'`, `'labels'`}, optional] Define what number is used as workshift's work time: workshift's duration (default) or the label. In the latter case, you need to use numbers as labels and it is up to you to interpret the values. See also *Work time* section in *Data Model*.

5.1.3 Example: Call center shifts with equal duration

Operators in a 24x7 call center work in three 8-hour shifts starting at 10:00, 18:00, and 02:00. For each operator one on-duty shift is followed by three off-duty shifts. Hence, four teams of operators are needed. They are designated as 'A', 'B', 'C', and 'D'.

```
>>> clnd = tb.Timeboard(base_unit_freq='8H',
...                     start='01 Oct 2017 02:00', end='05 Oct 2017 01:59',
...                     layout=['A', 'B', 'C', 'D'])
>>> print(clnd)
Timeboard of '8H': 2017-10-01 02:00 -> 2017-10-04 18:00

                 ws_ref ...                 end  label  on_duty
loc                     ...
0   2017-10-01 02:00:00 ...  2017-10-01 09:59:59      A     True
1   2017-10-01 10:00:00 ...  2017-10-01 17:59:59      B     True
2   2017-10-01 18:00:00 ...  2017-10-02 01:59:59      C     True
3   2017-10-02 02:00:00 ...  2017-10-02 09:59:59      D     True
4   2017-10-02 10:00:00 ...  2017-10-02 17:59:59      A     True
5   2017-10-02 18:00:00 ...  2017-10-03 01:59:59      B     True
6   2017-10-03 02:00:00 ...  2017-10-03 09:59:59      C     True
7   2017-10-03 10:00:00 ...  2017-10-03 17:59:59      D     True
8   2017-10-03 18:00:00 ...  2017-10-04 01:59:59      A     True
9   2017-10-04 02:00:00 ...  2017-10-04 09:59:59      B     True
10  2017-10-04 10:00:00 ...  2017-10-04 17:59:59      C     True
11  2017-10-04 18:00:00 ...  2017-10-05 01:59:59      D     True

# The "start" and "duration" columns have been omitted to fit the output
# to the page
```

There are two things in this example to point out.

First, to avoid the compound workshifts we use the 8-hour base unit but we need to align the base units with the workshifts, hence the frame starts at 02:00 o'clock.

Note: The duration of each workshift equals to one (base unit). Accordingly, work time of a workshift is also equal to one. To express workshift's duration or the work time in units of time, multiply it by the length of the base unit.

Second, all shifts are on duty because the default selector evaluates each label to True. It can be interpreted as the call center as a whole being always on duty. It is recommended to leave the default schedule as it is. In a later example,

we will see how it can be made useful.

To find out which workshifts are on duty for a team labeled with a particular symbol, you may add a schedule to the timeboard and supply the appropriate selector function:

```
>>> clnd.add_schedule(name='team_A', selector=lambda label: label=='A')
>>> print(clnd)
Timeboard of '8H': 2017-10-01 02:00 -> 2017-10-04 18:00

                 ws_ref ...                 end  label  on_duty   team_A
loc                     ...
0   2017-10-01 02:00:00 ... 2017-10-01 09:59:59      A     True     True
1   2017-10-01 10:00:00 ... 2017-10-01 17:59:59      B     True    False
2   2017-10-01 18:00:00 ... 2017-10-02 01:59:59      C     True    False
3   2017-10-02 02:00:00 ... 2017-10-02 09:59:59      D     True    False
4   2017-10-02 10:00:00 ... 2017-10-02 17:59:59      A     True     True
5   2017-10-02 18:00:00 ... 2017-10-03 01:59:59      B     True    False
6   2017-10-03 02:00:00 ... 2017-10-03 09:59:59      C     True    False
7   2017-10-03 10:00:00 ... 2017-10-03 17:59:59      D     True    False
8   2017-10-03 18:00:00 ... 2017-10-04 01:59:59      A     True     True
9   2017-10-04 02:00:00 ... 2017-10-04 09:59:59      B     True    False
10  2017-10-04 10:00:00 ... 2017-10-04 17:59:59      C     True    False
11  2017-10-04 18:00:00 ... 2017-10-05 01:59:59      D     True    False

# The "start" and "duration" columns have been omitted to fit the output
# to the page
```

5.2 Using *Organizer*

For most real-world scenarios a simple pattern of labels uniformly recurring across the whole timeboard is not sufficient for building a usable timeline. This is where `Organizer` comes into play.

`Organizer` tells how to partition the frame into chunks called 'spans' and how to structure each span into workshifts.

There are two mandatory parameters for an organizer. The first one is either *marks* or *marker* (but not both), it defines spans' boundaries. The second one is *structure*, it defines the structure of each span.

Below is an example of the organizer used to build a regular business calendar:

```
>>> weekly = tb.Organizer(marker='W', structure=[[1,1,1,1,1,0,0]])
```

An organizer is supplied to `Timeboard()` constructor in *layout* parameter instead of a pattern of labels which has been discussed in the previous section:

```
>>> clnd = tb.Timeboard(base_unit_freq='D',
...                     start='01 Oct 2017', end='12 Oct 2017',
...                     layout=weekly)
```

5.2.1 Parameters of *Organizer*

The first parameter of `Organizer()` - *marks* or *marker*, whichever is given, - tells where on the frame there will be marks designating the boundaries of spans. A mark is a point in time; the base unit containing this point in time will be the first base unit of a span.

If, for example, an organizer defines two marks, there will be three spans. The first span will begin on the first base unit of the frame and end on the base unit immediately preceding the unit containing the first mark. The second span will

begin on the base unit containing the first mark and end on the base unit immediately preceding the unit containing the second mark. The third span will begin on the base unit containing the second mark and end on the last base unit of the frame.

marks [Iterable] This is a list of explicit points in time which refer to the first base units of the spans.

A point in time is a *Timestamp*-like value (a *pandas.Timestamp*, or a string convertible to *Timestamp* (i.e. "10 Oct 2017 18:00"), or a *datetime* object). A point in time can be located anywhere within the base unit it refers to.

An empty *marks* list means that no partitioning is done, and the only span is the entire frame.

marker [str or Marker] You use *marker* to define the rule how to calculate the locations of marks rather than specify the explicit points in time as with *marks* parameter.

In simpler cases, the value of *marker* is a string representing a *pandas*-compatible calendar frequency (accepts the same kind of values as `base_unit_freq` of *Timeboard*; for example, `'W'` for weeks). The marks are set at the start times of the calendar periods, and as the result, the frame is partitioned into spans representing periods of the specified frequency.

Note that the first or the last span, or both may end up containing incomplete calendar periods. For example, the daily frame from 1 Oct 2017 through 12 Oct 2017 when partitioned with `marker='W'` produces three spans. The first span contains only 1 Oct 2017 as it was Sunday. The second span contains the full week from the Monday 2nd through the Sunday 8th of October. The last span consists of four days 9-12 of October which obviously do not form a complete week.

The parts of the "marker" calendar periods which fall outside the first and the last spans are called dangles. In our example the left dangle is the period from Monday 25 through Saturday 30 of September, and the right dangle is the period from Friday 13 through Sunday 15 of October:

```
               Mo  Tu  We  Th  Fr  Sa  Su
left dangle  : 25  26  27  28  29  30
span 0       :                           1   frame start='01 Oct 2017'
span 1       :  2   3   4   5   6   7   8
span 2       :  9  10  11  12              frame end='12 Oct 2017'
right dangle :                 13  14  15
```

The practical significance of dangles will be clarified shortly.

structure [Iterable] Each element of *structure* matches a span produced by partitioning: the first element of *structure* is applied to the first span, the second - to the second span, and so on. If *structure* gets exhausted, it starts over and iterates in cycles until the last span has been treated.

An element of *structure* can be one of the following:

- a pattern of labels : make each base unit a separate workshift, assign labels from the pattern;
- another `Organizer` : recursively organize the span into sub-spans;
- a single label : combine all base units of the span into a single compound workshift with the given label.

The following sections will provide examples of all these options.

Note: Under the hood, `layout=[1, 0, 0]` passed to `Timeboard()` is converted into `layout=Organizer(marks=[], structure=[[1, 0, 0]])`.

5.2.2 Example: Business day calendar

Note:

1. For the demonstration purposes, the timeboard is deliberately made short.
2. For the real-world usage, the holidays must be accounted for in the form of *amendments*. Here they are omitted for simplicity.

```
>>> weekly = tb.Organizer(marker='W', structure=[[1,1,1,1,1,0,0]])
>>> clnd = tb.Timeboard(base_unit_freq='D',
...                     start='01 Oct 2017', end='12 Oct 2017',
...                     layout=weekly)
```

In this example, the frame is partitioned into calendar weeks. This process produces three spans as shown in the previous section. The first span contains only Sunday 1 Oct 2017. The second span contains the full week from the Monday 2nd through the Sunday 8th of October. The last span consists of four days 9-12 of October.

The first element of *structure* is a list of values - a pattern. Therefore in the first span workshifts coincide with base units and receive labels from the pattern.

However, unlike the use of pattern directly in *layout* parameter of `Timeboard`, the first workshift of the span does not necessarily receive the first label of the pattern. If the span has a left dangle, the pattern starts with a shadow run through the length of the dangle. Only after that, it begins yielding labels for workshifts of the span. This approach can be viewed as if the dangle was attached to the first span to form the complete calendar period (in this example, a complete week) and then the pattern was applied from the start of the period but only those results (assigned labels) are retained that fall within the span. In this way, the workshift of October 1 receives the seventh label from the pattern, which is 0, after the first six labels have been shadow-assigned to the base units of the dangle.

The second span, a full week of October 2-8, is to be treated with the second element of *structure*. However, there is no second element. Consequently, *structure* is reenacted in cycles meaning that each span is treated with the first and the only element of the structure.

An interior span, such as the second span of this example, cannot have dangles. Therefore, the seven labels of the pattern are assigned in order to the seven workshifts of the second span.

The last, third span is again an incomplete week, but this time there is a right dangle. As patterns are currently applied only left to right, the presence of the right dangle does not produce any effect upon workshift labeling. The four workshifts of the third span receive the first four labels from the pattern.

The resulting calendar is printed below.

```
>>> print(clnd)
Timeboard of 'D': 2017-10-01 -> 2017-10-12

        ws_ref      start  duration        end  label  on_duty
loc
0   2017-10-01 2017-10-01         1 2017-10-01    0.0    False
1   2017-10-02 2017-10-02         1 2017-10-02    1.0     True
2   2017-10-03 2017-10-03         1 2017-10-03    1.0     True
3   2017-10-04 2017-10-04         1 2017-10-04    1.0     True
4   2017-10-05 2017-10-05         1 2017-10-05    1.0     True
5   2017-10-06 2017-10-06         1 2017-10-06    1.0     True
6   2017-10-07 2017-10-07         1 2017-10-07    0.0    False
7   2017-10-08 2017-10-08         1 2017-10-08    0.0    False
8   2017-10-09 2017-10-09         1 2017-10-09    1.0     True
9   2017-10-10 2017-10-10         1 2017-10-10    1.0     True
10  2017-10-11 2017-10-11         1 2017-10-11    1.0     True
11  2017-10-12 2017-10-12         1 2017-10-12    1.0     True
```

5.2.3 Example: Alternating week schedules

Consider a schedule of workshifts in a car dealership. A mechanic works on Monday, Tuesday, Saturday, and Sunday this week, and on Wednesday, Thursday, and Friday next week; then the bi-weekly cycle repeats.

```
>>> biweekly = tb.Organizer(marker='W',
...                         structure=[[1,1,0,0,0,1,1],[0,0,1,1,1,0,0]])
>>> clnd = tb.Timeboard(base_unit_freq='D',
...                     start='01 Oct 2017', end='22 Oct 2017',
...                     layout=biweekly)
>>> print(clnd)
Timeboard of 'D': 2017-10-01 -> 2017-10-22

        ws_ref      start  duration        end  label  on_duty
loc
0   2017-10-01 2017-10-01         1 2017-10-01    1.0     True
1   2017-10-02 2017-10-02         1 2017-10-02    0.0    False
2   2017-10-03 2017-10-03         1 2017-10-03    0.0    False
3   2017-10-04 2017-10-04         1 2017-10-04    1.0     True
4   2017-10-05 2017-10-05         1 2017-10-05    1.0     True
5   2017-10-06 2017-10-06         1 2017-10-06    1.0     True
6   2017-10-07 2017-10-07         1 2017-10-07    0.0    False
7   2017-10-08 2017-10-08         1 2017-10-08    0.0    False
8   2017-10-09 2017-10-09         1 2017-10-09    1.0     True
9   2017-10-10 2017-10-10         1 2017-10-10    1.0     True
10  2017-10-11 2017-10-11         1 2017-10-11    0.0    False
11  2017-10-12 2017-10-12         1 2017-10-12    0.0    False
12  2017-10-13 2017-10-13         1 2017-10-13    0.0    False
13  2017-10-14 2017-10-14         1 2017-10-14    1.0     True
14  2017-10-15 2017-10-15         1 2017-10-15    1.0     True
15  2017-10-16 2017-10-16         1 2017-10-16    0.0    False
16  2017-10-17 2017-10-17         1 2017-10-17    0.0    False
17  2017-10-18 2017-10-18         1 2017-10-18    1.0     True
18  2017-10-19 2017-10-19         1 2017-10-19    1.0     True
19  2017-10-20 2017-10-20         1 2017-10-20    1.0     True
20  2017-10-21 2017-10-21         1 2017-10-21    0.0    False
21  2017-10-22 2017-10-22         1 2017-10-22    0.0    False
```

5.2.4 Undersized and oversized patterns

A pattern supplied as an element of *structure* can be found undersized. It means that the pattern is shorter than the length of the span it is to be applied to. In this case the pattern will be reenacted in cycles until the full length of the span has been covered.

If at the same time, the span has a left dangle associated with it, then the approach is consistent with the one described in the previous section. The dangle is attached to the beginning of the span. Then the pattern is run in cycles over the combined dangle-and-span retaining only those labels that belong to the span.

The example below illustrates the behavior of undersized patterns. It shows the calendar of activities happening on odd days of the week.

```
>>> weekly = tb.Organizer(marker='W', structure=[[1,0]])
>>> clnd = tb.Timeboard(base_unit_freq='D',
...                     start='01 Oct 2017', end='12 Oct 2017',
...                     layout=weekly)
>>> print(clnd)
```

(continues on next page)

(continued from previous page)

```
Timeboard of 'D': 2017-10-01 -> 2017-10-12

        ws_ref      start  duration        end  label  on_duty
loc
0   2017-10-01 2017-10-01         1 2017-10-01    1.0     True
1   2017-10-02 2017-10-02         1 2017-10-02    1.0     True
2   2017-10-03 2017-10-03         1 2017-10-03    0.0    False
3   2017-10-04 2017-10-04         1 2017-10-04    1.0     True
4   2017-10-05 2017-10-05         1 2017-10-05    0.0    False
5   2017-10-06 2017-10-06         1 2017-10-06    1.0     True
6   2017-10-07 2017-10-07         1 2017-10-07    0.0    False
7   2017-10-08 2017-10-08         1 2017-10-08    1.0     True
8   2017-10-09 2017-10-09         1 2017-10-09    1.0     True
9   2017-10-10 2017-10-10         1 2017-10-10    0.0    False
10  2017-10-11 2017-10-11         1 2017-10-11    1.0     True
11  2017-10-12 2017-10-12         1 2017-10-12    0.0    False
```

Note that the first of October receives label `1` after the pattern `[1, 0]` has completed three shadow cycles over the six-day dangle.

If the pattern is oversized, meaning it is longer than the span, the excess labels are ignored. Should the same pattern be applied to another span in the next cycle through *structure*, the labeling restarts from the beginning of the pattern.

5.3 Recursive organizing

A small museum's schedule is seasonal. In winter (November through April) the museum is open only on Wednesdays and Thursdays, but in summer (May through October) the museum works every day except Monday.

```
>>> winter = tb.Organizer(marker='W', structure=[[0,0,1,1,0,0,0]])
>>> summer = tb.Organizer(marker='W', structure=[[0,1,1,1,1,1,1]])
>>> seasonal = tb.Organizer(marker='6M', structure=[winter, summer])
>>> clnd = tb.Timeboard(base_unit_freq='D',
...                     start='01 Nov 2015', end='31 Oct 2017',
...                     layout=seasonal)
```

In this example there are two levels of organizers.

On the outer level *seasonal* organizer partitions the frame into spans of 6 months each. The spans represent, alternatively, winter and summer seasons. The *structure* of this organizer, instead of patterns of labels, contains other organizers. These inner level organizers, named *winter* and *summer*, are applied, in turns, to the spans produced by *seasonal* organizer as if they were whole frames.

On the inner level, each season is partitioned into weeks by *winter* or *summer* organizer correspondingly. As the result, workshifts within the weeks of each season receive labels from the patterns specific for the seasons.

```
>>> print(clnd)
Timeboard of 'D': 2015-11-01 -> 2017-10-31

        ws_ref      start  duration        end  label  on_duty
loc
0   2015-11-01 2015-11-01         1 2015-11-01    0.0    False
1   2015-11-02 2015-11-02         1 2015-11-02    0.0    False
2   2015-11-03 2015-11-03         1 2015-11-03    0.0    False
3   2015-11-04 2015-11-04         1 2015-11-04    1.0     True
4   2015-11-05 2015-11-05         1 2015-11-05    1.0     True
```

(continues on next page)

(continued from previous page)

```
5   2015-11-06 2015-11-06          1 2015-11-06    0.0    False
6   2015-11-07 2015-11-07          1 2015-11-07    0.0    False
7   2015-11-08 2015-11-08          1 2015-11-08    0.0    False
8   2015-11-09 2015-11-09          1 2015-11-09    0.0    False
9   2015-11-10 2015-11-10          1 2015-11-10    0.0    False
10  2015-11-11 2015-11-11          1 2015-11-11    1.0     True
11  2015-11-12 2015-11-12          1 2015-11-12    1.0     True
12  2015-11-13 2015-11-13          1 2015-11-13    0.0    False
13  2015-11-14 2015-11-14          1 2015-11-14    0.0    False
14  2015-11-15 2015-11-15          1 2015-11-15    0.0    False
...        ...        ...        ...        ...    ...      ...
715 2017-10-16 2017-10-16          1 2017-10-16    0.0    False
716 2017-10-17 2017-10-17          1 2017-10-17    1.0     True
717 2017-10-18 2017-10-18          1 2017-10-18    1.0     True
718 2017-10-19 2017-10-19          1 2017-10-19    1.0     True
719 2017-10-20 2017-10-20          1 2017-10-20    1.0     True
720 2017-10-21 2017-10-21          1 2017-10-21    1.0     True
721 2017-10-22 2017-10-22          1 2017-10-22    1.0     True
722 2017-10-23 2017-10-23          1 2017-10-23    0.0    False
723 2017-10-24 2017-10-24          1 2017-10-24    1.0     True
724 2017-10-25 2017-10-25          1 2017-10-25    1.0     True
725 2017-10-26 2017-10-26          1 2017-10-26    1.0     True
726 2017-10-27 2017-10-27          1 2017-10-27    1.0     True
727 2017-10-28 2017-10-28          1 2017-10-28    1.0     True
728 2017-10-29 2017-10-29          1 2017-10-29    1.0     True
729 2017-10-30 2017-10-30          1 2017-10-30    0.0    False
730 2017-10-31 2017-10-31          1 2017-10-31    1.0     True

[731 rows x 6 columns]
```

There may be any number of recursion levels for organizers.

5.4 Using *Marker*

The museum's schedule discussed in the previous section is contrived in that each season is exactly 6 months long. If, for example, the summer season began on the 1st of May and ended on the 15th of September, we could not construct the timeline by merely partitioning the frame with calendar periods.

More sophisticated partitioning is achieved with the tool called `Marker`. For example, the marks for seasons starting annually on May 1 and Sep 16 are set by:

```
tb.Marker(each='A', at=[{'months':4}, {'months':8, 'days':15}])
```

`Marker()` constructor takes one mandatory parameter, *each*, but the real power comes with the use of parameter *at*.

each [str] *pandas*-compatible calendar frequency (accepts the same kind of values as `base_unit_freq` of *Timeboard*).

at [list of dict, optional] This is an iterable of dictionaries. Each dictionary specifies a time offset using such keywords as `'months'`, `'days'`, `'hours'`, etc.

For each calendar period of frequency *each*, we obtain candidate marks by adding offsets from *at* list to the start time of the period. After that we retain only those candidates that fall within the period (and, obviously, within the frame) - these points become the marks.

The expression in the above example:

```
tb.Marker(each='A', at=[{'months':4}, {'months':8, 'days':15}])
```

means:

```
there are two marks per year;
to get the first mark add 4 months to the start of each year;
to get the second mark add 8 months and 15 days to the start of the same year.
```

As a result, the frame is partitioned into spans starting on the 1st of May and on the 16th of September of each year provided that these dates are within the frame bounds.

An instance of `Marker` is passed to `Organizer()` constructor as the value of *marker* parameter instead of a simple calendar frequency.

5.4.1 Example: Seasonal schedule

Here comes a more realistic schedule for the small museum. In winter (September 16 through April 30) the museum is open only on Wednesdays and Thursdays, but in summer (May 1 through September 15) the museum works every day except Monday.

```
>>> winter = tb.Organizer(marker='W', structure=[[0,0,1,1,0,0,0]])
>>> summer = tb.Organizer(marker='W', structure=[[0,1,1,1,1,1,1]])
>>> seasons =  tb.Marker(each='A',
...                      at=[{'months':4}, {'months':8, 'days':15}])
>>> seasonal = tb.Organizer(marker=seasons,
...                         structure=[winter, summer])
>>> clnd = tb.Timeboard(base_unit_freq='D',
...                     start='01 Jan 2015', end='31 Dec 2017',
...                     layout=seasonal)
```

As the timeboard is too long, we will print only intervals around the marks.

```
>>> print(clnd(('20 Apr 2017','10 May 2017')))
Interval((840, 860)): 'D' at 2017-04-20 -> 'D' at 2017-05-10 [21]

        ws_ref      start  duration        end  label  on_duty
loc
840 2017-04-20 2017-04-20         1 2017-04-20    1.0     True
841 2017-04-21 2017-04-21         1 2017-04-21    0.0    False
842 2017-04-22 2017-04-22         1 2017-04-22    0.0    False
843 2017-04-23 2017-04-23         1 2017-04-23    0.0    False
844 2017-04-24 2017-04-24         1 2017-04-24    0.0    False
845 2017-04-25 2017-04-25         1 2017-04-25    0.0    False
846 2017-04-26 2017-04-26         1 2017-04-26    1.0     True
847 2017-04-27 2017-04-27         1 2017-04-27    1.0     True
848 2017-04-28 2017-04-28         1 2017-04-28    0.0    False
849 2017-04-29 2017-04-29         1 2017-04-29    0.0    False
850 2017-04-30 2017-04-30         1 2017-04-30    0.0    False
851 2017-05-01 2017-05-01         1 2017-05-01    0.0    False
852 2017-05-02 2017-05-02         1 2017-05-02    1.0     True
853 2017-05-03 2017-05-03         1 2017-05-03    1.0     True
854 2017-05-04 2017-05-04         1 2017-05-04    1.0     True
855 2017-05-05 2017-05-05         1 2017-05-05    1.0     True
856 2017-05-06 2017-05-06         1 2017-05-06    1.0     True
857 2017-05-07 2017-05-07         1 2017-05-07    1.0     True
858 2017-05-08 2017-05-08         1 2017-05-08    0.0    False
```

(continues on next page)

(continued from previous page)

```
859 2017-05-09 2017-05-09         1 2017-05-09    1.0     True
860 2017-05-10 2017-05-10         1 2017-05-10    1.0     True

>>> print(clnd(('04 Sep 2017','24 Sep 2017')))
Interval((977, 997)): 'D' at 2017-09-04 -> 'D' at 2017-09-24 [21]

        ws_ref      start  duration        end  label  on_duty
loc
977 2017-09-04 2017-09-04         1 2017-09-04    0.0    False
978 2017-09-05 2017-09-05         1 2017-09-05    1.0     True
979 2017-09-06 2017-09-06         1 2017-09-06    1.0     True
980 2017-09-07 2017-09-07         1 2017-09-07    1.0     True
981 2017-09-08 2017-09-08         1 2017-09-08    1.0     True
982 2017-09-09 2017-09-09         1 2017-09-09    1.0     True
983 2017-09-10 2017-09-10         1 2017-09-10    1.0     True
984 2017-09-11 2017-09-11         1 2017-09-11    0.0    False
985 2017-09-12 2017-09-12         1 2017-09-12    1.0     True
986 2017-09-13 2017-09-13         1 2017-09-13    1.0     True
987 2017-09-14 2017-09-14         1 2017-09-14    1.0     True
988 2017-09-15 2017-09-15         1 2017-09-15    1.0     True
989 2017-09-16 2017-09-16         1 2017-09-16    0.0    False
990 2017-09-17 2017-09-17         1 2017-09-17    0.0    False
991 2017-09-18 2017-09-18         1 2017-09-18    0.0    False
992 2017-09-19 2017-09-19         1 2017-09-19    0.0    False
993 2017-09-20 2017-09-20         1 2017-09-20    1.0     True
994 2017-09-21 2017-09-21         1 2017-09-21    1.0     True
995 2017-09-22 2017-09-22         1 2017-09-22    0.0    False
996 2017-09-23 2017-09-23         1 2017-09-23    0.0    False
997 2017-09-24 2017-09-24         1 2017-09-24    0.0    False
```

If *at* parameter is not given or is an empty list, the marks are placed at the start times of the calendar periods specified by *each*.

Note: Under the hood, `marker='x'` passed to `Organizer()` is converted into `marker=Marker(each='x')`.

It should be emphasized that in the presence of non-empty *at* list the frame is partitioned on the *each* period boundary only if it is explicitly defined in *at* in the form of the zero offset (i.e. `at=[{'days':0}, ... ]`).

If *at* list is non-empty but its processing has not produced any valid marks, no partitioning occurs.

Note that now we do not have to align the start of the frame with the start of a season. However, we must make sure that, if the frame starts in winter, then the first element in the structure of *seasonal* organizer is the organizer that is responsible for winter and vice versa.

5.4.2 Using parameter *how*

`Marker()` constructor has the third parameter *how* which defines the interpretation of keyword arguments provided in *at* list:

Value of *how*	Interpretation of keyword arguments in *at*
'from_start_of_each'	Keyword arguments define an offset from the beginning of *each* period. Acceptable keyword arguments are `'seconds'`, `'minutes'`, `'hours'`, `'days'`, `'weeks'`, `'months'`, `'years'`. Example: `at=[{'days':0}, {'days':1, 'hours':2}]` (the first mark is at the start of the period, the second is in 1 day and 2 hours from the start of the period).
'from_easter_western'	Keyword arguments define an offset from the day of Western Easter. Acceptable arguments are the same as above.
'from_easter_orthodox'	Keyword arguments define an offset from the day of Orthodox Easter. Acceptable arguments are the same as above.
'nth_weekday_of_month'	Keywords arguments refer to N-th weekday of M-th month from the start of *each* period. Acceptable keywords are: • **`'month'`** [1..12] 1 is for the first month (such as January for the annual frequency). • **`'weekday'`** [1..7] 1 is for Monday, 7 is for Sunday. • **`'week'`** [-5..-1, 1..5] -1 is for the last and 1 is for the first occurrence of the weekday in the month. Zero is not allowed. • **`'shift'`** [int, optional, default 0] An offset in days from the weekday found. Example: `at=[{'month':5, 'weekday':7, 'week':-1}]` (the last Sunday of the 5th month)

The options `'from_easter_western'` and `'from_easter_orthodox'` assume the same format of *at* keywords as the default option `'from_start_of_each'` which has been explored in the previous section. The difference is that the offset now may be negative. For example,

```
tb.Marker(each='A', at=[{'days': -2}], how='from_easter_western')
```

sets marks at 00:00 on Good Fridays.

5.4.3 Example: Seasons turning on n-th weekday of month

The museum's summer season starts on a Tuesday after the first Monday in May and ends on the last Sunday in September. During summer the museum is open every day except Monday; during winter it is open on Wednesdays and Thursdays only.

```
>>> winter = tb.Organizer(marker='W', structure=[[0,0,1,1,0,0,0]])
>>> summer = tb.Organizer(marker='W', structure=[[0,1,1,1,1,1,1]])
>>> seasons =  tb.Marker(each='A',
...                      at=[{'month':5, 'weekday':1, 'week':1, 'shift':1},
...                          {'month':9, 'weekday':7, 'week':-1}],
...                      how='nth_weekday_of_month')
>>> seasonal = tb.Organizer(marker=seasons,
...                         structure=[winter, summer])
```

(continues on next page)

(continued from previous page)

```
>>> clnd = tb.Timeboard(base_unit_freq='D',
...                     start='01 Jan 2012', end='31 Dec 2015',
...                     layout=seasonal)

>>> print(clnd(('30 Apr 2012','15 May 2012')))
Interval((120, 135)): 'D' at 2012-04-30 -> 'D' at 2012-05-15 [16]

        ws_ref      start  duration        end  label  on_duty
loc
120 2012-04-30 2012-04-30         1 2012-04-30    0.0    False
121 2012-05-01 2012-05-01         1 2012-05-01    0.0    False
122 2012-05-02 2012-05-02         1 2012-05-02    1.0     True
123 2012-05-03 2012-05-03         1 2012-05-03    1.0     True
124 2012-05-04 2012-05-04         1 2012-05-04    0.0    False
125 2012-05-05 2012-05-05         1 2012-05-05    0.0    False
126 2012-05-06 2012-05-06         1 2012-05-06    0.0    False
127 2012-05-07 2012-05-07         1 2012-05-07    0.0    False
128 2012-05-08 2012-05-08         1 2012-05-08    1.0     True
129 2012-05-09 2012-05-09         1 2012-05-09    1.0     True
130 2012-05-10 2012-05-10         1 2012-05-10    1.0     True
131 2012-05-11 2012-05-11         1 2012-05-11    1.0     True
132 2012-05-12 2012-05-12         1 2012-05-12    1.0     True
133 2012-05-13 2012-05-13         1 2012-05-13    1.0     True
134 2012-05-14 2012-05-14         1 2012-05-14    0.0    False
135 2012-05-15 2012-05-15         1 2012-05-15    1.0     True
```

Note that 1 May 2012 was Tuesday, so the Tuesday after the first Monday was 8 May 2012. The last Sunday in September 2012 was the 30th.

```
>>> print(clnd(('23 Sep 2012','07 Oct 2012')))
Interval((266, 280)): 'D' at 2012-09-23 -> 'D' at 2012-10-07 [15]

        ws_ref      start  duration        end  label  on_duty
loc
266 2012-09-23 2012-09-23         1 2012-09-23    1.0     True
267 2012-09-24 2012-09-24         1 2012-09-24    0.0    False
268 2012-09-25 2012-09-25         1 2012-09-25    1.0     True
269 2012-09-26 2012-09-26         1 2012-09-26    1.0     True
270 2012-09-27 2012-09-27         1 2012-09-27    1.0     True
271 2012-09-28 2012-09-28         1 2012-09-28    1.0     True
272 2012-09-29 2012-09-29         1 2012-09-29    1.0     True
273 2012-09-30 2012-09-30         1 2012-09-30    0.0    False
274 2012-10-01 2012-10-01         1 2012-10-01    0.0    False
275 2012-10-02 2012-10-02         1 2012-10-02    0.0    False
276 2012-10-03 2012-10-03         1 2012-10-03    1.0     True
277 2012-10-04 2012-10-04         1 2012-10-04    1.0     True
278 2012-10-05 2012-10-05         1 2012-10-05    0.0    False
279 2012-10-06 2012-10-06         1 2012-10-06    0.0    False
280 2012-10-07 2012-10-07         1 2012-10-07    0.0    False
```

5.5 Using pattern with memory

A school administrator's work schedule is 2 days working followed by 3 days off, with a recess from 14 Jul to 31 Aug every year:

```
>>> year = tb.Marker(each='A',
...                  at=[{'months':6, 'days':13}, {'months':8}])
>>> annually = tb.Organizer(marker=year,
...                         structure=[[1,1,1,0,0],[-1]])
>>> clnd = tb.Timeboard(base_unit_freq='D',
...                     start='01 Jan 2016', end='31 Dec 2017',
...                     layout=annually,
...                     default_selector=lambda label: label>0)
```

The days of the recess are labeled with `-1` to differentiate them from the regular days off. The selector for the default schedule has been adjusted accordingly.

```
>>> print(clnd(('07 Jul 2016','17 Jul 2016')))
Interval((188, 198)): 'D' at 2016-07-07 -> 'D' at 2016-07-17 [11]

        ws_ref      start  duration        end  label  on_duty
loc
188 2016-07-07 2016-07-07         1 2016-07-07    1.0     True
189 2016-07-08 2016-07-08         1 2016-07-08    1.0     True
190 2016-07-09 2016-07-09         1 2016-07-09    1.0     True
191 2016-07-10 2016-07-10         1 2016-07-10    0.0    False
192 2016-07-11 2016-07-11         1 2016-07-11    0.0    False
193 2016-07-12 2016-07-12         1 2016-07-12    1.0     True
194 2016-07-13 2016-07-13         1 2016-07-13    1.0     True
195 2016-07-14 2016-07-14         1 2016-07-14   -1.0    False
196 2016-07-15 2016-07-15         1 2016-07-15   -1.0    False
197 2016-07-16 2016-07-16         1 2016-07-16   -1.0    False
198 2016-07-17 2016-07-17         1 2016-07-17   -1.0    False

>>> print(clnd(('27 Aug 2016','06 Sep 2016')))
Interval((239, 249)): 'D' at 2016-08-27 -> 'D' at 2016-09-06 [11]

        ws_ref      start  duration        end  label  on_duty
loc
239 2016-08-27 2016-08-27         1 2016-08-27   -1.0    False
240 2016-08-28 2016-08-28         1 2016-08-28   -1.0    False
241 2016-08-29 2016-08-29         1 2016-08-29   -1.0    False
242 2016-08-30 2016-08-30         1 2016-08-30   -1.0    False
243 2016-08-31 2016-08-31         1 2016-08-31   -1.0    False
244 2016-09-01 2016-09-01         1 2016-09-01    1.0     True
245 2016-09-02 2016-09-02         1 2016-09-02    1.0     True
246 2016-09-03 2016-09-03         1 2016-09-03    1.0     True
247 2016-09-04 2016-09-04         1 2016-09-04    0.0    False
248 2016-09-05 2016-09-05         1 2016-09-05    0.0    False
249 2016-09-06 2016-09-06         1 2016-09-06    1.0     True
```

Note that the working period before the recess has ended mid-cycle: the administrator has checked out only two (Jul 12 and Jul 13) of five days forming a complete cycle. The period after the recess started afresh with three working days followed by two days off. This is the expected behavior as *Organizer* applies the next element of *structure* to the next span without knowledge of any previous invocations of this element.

However, if you wish to retain the flow of administrator's schedule as if it was uninterrupted by the recess, you may employ `RememberingPattern`. This class creates a pattern which memorizes its state from previous invocations across all organizers. It takes only one parameter - an iterable of labels.

```
>>> work_cycle = tb.RememberingPattern([1,1,1,0,0])
>>> year = tb.Marker(each='A',
```

(continues on next page)

(continued from previous page)

```
...                   at=[{'months':6, 'days':13}, {'months':8}])
>>> annually = tb.Organizer(marker=year,
...                         structure=[work_cycle,[-1]])
>>> clnd = tb.Timeboard(base_unit_freq='D',
...                     start='01 Jan 2016', end='31 Dec 2017',
                        layout=annually,
                        default_selector=lambda x: x>0)

>>> print(clnd(('07 Jul 2016','17 Jul 2016')))
Interval((188, 198)): 'D' at 2016-07-07 -> 'D' at 2016-07-17 [11]

        ws_ref      start  duration        end  label  on_duty
loc
188 2016-07-07 2016-07-07         1 2016-07-07    1.0     True
189 2016-07-08 2016-07-08         1 2016-07-08    1.0     True
190 2016-07-09 2016-07-09         1 2016-07-09    1.0     True
191 2016-07-10 2016-07-10         1 2016-07-10    0.0    False
192 2016-07-11 2016-07-11         1 2016-07-11    0.0    False
193 2016-07-12 2016-07-12         1 2016-07-12    1.0     True
194 2016-07-13 2016-07-13         1 2016-07-13    1.0     True
195 2016-07-14 2016-07-14         1 2016-07-14   -1.0    False
196 2016-07-15 2016-07-15         1 2016-07-15   -1.0    False
197 2016-07-16 2016-07-16         1 2016-07-16   -1.0    False
198 2016-07-17 2016-07-17         1 2016-07-17   -1.0    False

>>> print(clnd(('27 Aug 2016','08 Sep 2016')))
Interval((239, 251)): 'D' at 2016-08-27 -> 'D' at 2016-09-08 [13]

        ws_ref      start  duration        end  label  on_duty
loc
239 2016-08-27 2016-08-27         1 2016-08-27   -1.0    False
240 2016-08-28 2016-08-28         1 2016-08-28   -1.0    False
241 2016-08-29 2016-08-29         1 2016-08-29   -1.0    False
242 2016-08-30 2016-08-30         1 2016-08-30   -1.0    False
243 2016-08-31 2016-08-31         1 2016-08-31   -1.0    False
244 2016-09-01 2016-09-01         1 2016-09-01    1.0     True
245 2016-09-02 2016-09-02         1 2016-09-02    0.0    False
246 2016-09-03 2016-09-03         1 2016-09-03    0.0    False
247 2016-09-04 2016-09-04         1 2016-09-04    1.0     True
248 2016-09-05 2016-09-05         1 2016-09-05    1.0     True
249 2016-09-06 2016-09-06         1 2016-09-06    1.0     True
250 2016-09-07 2016-09-07         1 2016-09-07    0.0    False
251 2016-09-08 2016-09-08         1 2016-09-08    0.0    False
```

The period after the recess started with a shortened cycle consisting of one working day (Sep 1) followed by two days off (Sep 2 and 3). These days were "carried over" from the period before recess to complete the cycle started on the 12th of July.

5.6 Adjusting labels for work time

In the above examples with daily workshifts, the actual work time takes only a part of the workshift (that is, a part of the 24 hour day). If the amount of the work time varies between on-duty workshifts (for example, Friday's working hours in the office are shorter), these variations cannot be inferred from workshift's duration which is always equal to one (day). Therefore, you have to use labels as the source of the information about work time.

So far we have used simplistic labeling: `0` for an off-duty day and `1` for an on-duty day. To make work time measuring possible, the labeling scheme must be changed. The labels for off-duty days remain zero but the labels for on-duty days will be equal to the workshift's work time (presumably, measured in hours but this is up to the user). There is no need to change the selector. Yet you must add `worktime_source='labels'` to the parameters of timeboard.

The adjusted timeboard of the museum accounts for short days in winter and longer days in summer with extended working hours on Sunday and Mondays. The changes are in the first two lines and in the last.

```
>>> winter = tb.Organizer(marker='W', structure=[[0,0,6,6,0,0,0]])
>>> summer = tb.Organizer(marker='W', structure=[[0,8,8,8,8,10,10]])
>>> seasons =  tb.Marker(each='A',
...                       at=[{'month':5, 'weekday':1, 'week':1, 'shift':1},
...                           {'month':9, 'weekday':7, 'week':-1}],
...                       how='nth_weekday_of_month')
>>> seasonal = tb.Organizer(marker=seasons,
...                         structure=[winter, summer])
>>> clnd = tb.Timeboard(base_unit_freq='D',
...                     start='01 Jan 2012', end='31 Dec 2015',
...                     layout=seasonal,
...                     worktime_source='labels')
```

5.7 Workshifts of varying length

Let us modify the schedule of the 24x7 call center. Now the call center's staff operate in shifts of varying length: 08:00 to 18:00 (10 hours), 18:00 to 02:00 (8 hours), and 02:00 to 08:00 (6 hours). An operator's schedule consists of one on-duty shift followed by three off-duty shifts. Hence, four teams of operators are needed. They are designated as 'A', 'B', 'C', and 'D'.

To accommodate periods of varying length you need to use compound workshifts. A compound workshift consists of several base units.

Note: See also *Compound Workshifts* section in *Data Model* for the discussion about why and when you need compound workshifts.

Compound workshift is created from a span when a corresponding element of *structure* is neither a pattern nor an organizer. The value of such element is considered the label for the compound workshift. The workshift consists of all base units of the corresponding span.

```
>>> day_parts = tb.Marker(each='D',
...                       at=[{'hours':2}, {'hours':8}, {'hours':18}])
>>> shifts = tb.Organizer(marker=day_parts, structure=['A', 'B', 'C', 'D'])
>>> clnd = tb.Timeboard(base_unit_freq='H',
...                     start='02 Oct 2017 08:00', end='07 Oct 2017 01:59',
...                     layout=shifts)
>>> clnd.add_schedule(name='team_A', selector=lambda label: label=='A')

>>>print(clnd)
Timeboard of 'H': 2017-10-02 08:00 -> 2017-10-07 01:00

                ws_ref ... dur.                 end  label on_duty  team_A
loc                    ...
0   2017-10-02 08:00:00 ...   10 2017-10-02 17:59:59      A    True    True
1   2017-10-02 18:00:00 ...    8 2017-10-03 01:59:59      B    True   False
2   2017-10-03 02:00:00 ...    6 2017-10-03 07:59:59      C    True   False
```

(continues on next page)

(continued from previous page)

```
3   2017-10-03 08:00:00 ...   10 2017-10-03 17:59:59      D    True   False
4   2017-10-03 18:00:00 ...    8 2017-10-04 01:59:59      A    True    True
5   2017-10-04 02:00:00 ...    6 2017-10-04 07:59:59      B    True   False
6   2017-10-04 08:00:00 ...   10 2017-10-04 17:59:59      C    True   False
7   2017-10-04 18:00:00 ...    8 2017-10-05 01:59:59      D    True   False
8   2017-10-05 02:00:00 ...    6 2017-10-05 07:59:59      A    True    True
9   2017-10-05 08:00:00 ...   10 2017-10-05 17:59:59      B    True   False
10  2017-10-05 18:00:00 ...    8 2017-10-06 01:59:59      C    True   False
11  2017-10-06 02:00:00 ...    6 2017-10-06 07:59:59      D    True   False
12  2017-10-06 08:00:00 ...   10 2017-10-06 17:59:59      A    True    True
13  2017-10-06 18:00:00 ...    8 2017-10-07 01:59:59      B    True   False

# The "start" column has been omitted and "duration" squeezed to fit
# the output to the page
```

5.7.1 Example: Call center closing on weekends

We proceed with elaborating upon the schedule of the call center. In this example we employ all the tools we have at hand.

Suppose that the call center is located in Europe and supports traders doing business on stock exchanges around the world. Since markets are closed on Saturdays and Sundays, there is no need to staff the call center from 2:00 on Saturday (New York closes) to 2:00 on Monday (Tokyo opens).

To adjust the timeboard to this specific schedule, we need to modify the timeline in such a way that it takes into account days of the week. This job is carried out by marker *week* and organizer *weekly*.

Moreover, we will need a `RememberingPattern` to ensure that the order of the team rotation is not disrupted by weekends. Without `RememberingPattern` the first shift of each week will be always assigned to team A regardless of what team has staffed the last shift on the previous week.

```
>>> shifts_order = tb.RememberingPattern(['A', 'B', 'C', 'D'])
>>> day_parts = tb.Marker(each='D',
...                       at=[{'hours':2}, {'hours':8}, {'hours':18}])
>>> shifts = tb.Organizer(marker=day_parts, structure=shifts_order)
>>> week = tb.Marker(each='W',
...                  at=[{'days':0, 'hours':2}, {'days':5, 'hours':2}])
>>> weekly = tb.Organizer(marker=week, structure=[0, shifts])
>>> clnd = tb.Timeboard(base_unit_freq='H',
...                     start='02 Oct 2017 00:00', end='10 Oct 2017 01:59',
...                     layout=weekly)
>>> clnd.add_schedule(name='team_A', selector=lambda label: label=='A')
>>>
>>> print(clnd)
Timeboard of 'H': 2017-10-02 00:00 -> 2017-10-10 01:00

                 ws_ref ... dur.                 end label  on_duty  team_A
loc                     ...
0   2017-10-02 00:00:00 ...    2 2017-10-02 01:59:59     0    False   False
1   2017-10-02 02:00:00 ...    6 2017-10-02 07:59:59     A     True    True
2   2017-10-02 08:00:00 ...   10 2017-10-02 17:59:59     B     True   False
3   2017-10-02 18:00:00 ...    8 2017-10-03 01:59:59     C     True   False
4   2017-10-03 02:00:00 ...    6 2017-10-03 07:59:59     D     True   False
5   2017-10-03 08:00:00 ...   10 2017-10-03 17:59:59     A     True    True
6   2017-10-03 18:00:00 ...    8 2017-10-04 01:59:59     B     True   False
```

(continues on next page)

(continued from previous page)

```
7   2017-10-04 02:00:00 ...    6 2017-10-04 07:59:59     C     True   False
8   2017-10-04 08:00:00 ...   10 2017-10-04 17:59:59     D     True   False
9   2017-10-04 18:00:00 ...    8 2017-10-05 01:59:59     A     True    True
10  2017-10-05 02:00:00 ...    6 2017-10-05 07:59:59     B     True   False
11  2017-10-05 08:00:00 ...   10 2017-10-05 17:59:59     C     True   False
12  2017-10-05 18:00:00 ...    8 2017-10-06 01:59:59     D     True   False
13  2017-10-06 02:00:00 ...    6 2017-10-06 07:59:59     A     True    True
14  2017-10-06 08:00:00 ...   10 2017-10-06 17:59:59     B     True   False
15  2017-10-06 18:00:00 ...    8 2017-10-07 01:59:59     C     True   False
16  2017-10-07 02:00:00 ...   48 2017-10-09 01:59:59     0    False   False
17  2017-10-09 02:00:00 ...    6 2017-10-09 07:59:59     D     True   False
18  2017-10-09 08:00:00 ...   10 2017-10-09 17:59:59     A     True    True
19  2017-10-09 18:00:00 ...    8 2017-10-10 01:59:59     B     True   False

# The "start" column has been omitted and "duration" squeezed to fit
# the output to the page
```

Label `0` denotes the periods of time when the call center is closed: during first two hours of Monday 2 October, and from 02:00 on Saturday 7 October through 01:59 on Monday 9 October.

The default schedule ('on_duty') now becomes informative as it shows the schedule of the call center as a whole. We also added a schedule for team 'A'. For the practical use you will want to add schedules for the other shifts but this is not the point of this example.

The first week ends on shift 'C', and the next week starts with shift 'D', so the order of shifts is preserved which is an essential requirement for this timeboard.

To enable measurements of work time no adjustments of the timeboard's parameters are necessary. By default, the work time is assumed to be equal to workshift's duration. This is the case in this timeboard.

5.8 Caveats

5.8.1 Not all *Marker* frequencies are valid

Currently, *UnacceptablePeriodError* is raised for some combinations of base unit frequency and *Marker* frequency which may result in one base unit belonging to different adjacent calendar periods marked by the *Marker*.

Base unit is not a subperiod

Organizing fails when base unit is not a natural subperiod of the period used by *Marker*, for example:

```
>>> org = tb.Organizer(marker='M', structure=[[1]])
>>> clnd = tb.Timeboard(base_unit_freq='W',
...                     start='01 Oct 2017', end='31 Dec 2017',
...                     layout=org)
---------------------------------------------------------------------------
UnacceptablePeriodError                    Traceback (most recent call last)
  ...
UnacceptablePeriodError: Ambiguous organizing: W is not a subperiod of M
```

Indeed, a week may start in one month, and end in another, therefore it is not obvious to which span such a base unit should belong.

Actually, week is the only such irregular calendar frequency which is not a subperiod of anything. However, it is unlikely that week-sized base units will be a common occurrence in practice.

Base unit of multiplied frequency

Organizing fails when the base unit has a multiplied frequency (i.e. `'2H'`) **and** the period used by Marker is based on a **different** frequency.

This problem is less obvious and may manifest itself in some practical cases.

Consider first the legitimate code:

```
>>> org = tb.Organizer(marker='W', structure=[[1]])
>>> clnd = tb.Timeboard(base_unit_freq='D',
...                     start='02 Oct 2017', end='15 Oct 2017',
...                     layout=org)
>>> print(clnd)
Timeboard of 'D': 2017-10-02 -> 2017-10-15
  ...
```

Now change base unit frequency from `'D'` to `'24H'`:

```
>>> org = tb.Organizer(marker='W', structure=[[1]])
>>> clnd = tb.Timeboard(base_unit_freq='24H',
...                     start='02 Oct 2017', end='15 Oct 2017',
...                     layout=org)
---------------------------------------------------------------------------
UnacceptablePeriodError                   Traceback (most recent call last)
  ...
UnacceptablePeriodError: Ambiguous organizing: 24H is not a subperiod of W
```

It failed for the following reason. A period of frequency `'D'` always starts at 00:00 of a calendar day and thus is guaranteed to be entirely within some week. A period of frequency `'24H'` is guaranteed to start at the beginning of some hour but this hour is not necessarily a midnight. For example, a `'24H'` period *may* start at 20:00 of a Sunday, therefore its first four hours will fall into one week, and the rest - into another.

Example of a real-life case impacted by this issue: workshifts beginning or ending at half past hour. You cannot use `'30T'` (30 minutes) as a period for base units because you will have to organize the base units into shifts (presumably, with `each='D'`). A workaround is to select `'T'` as the base unit frequency. The side-effects are the slower performance and the rise in memory consumption.

While you may ensure that the base units start at midnights, `timeboard` is not yet able to handle base unit alignments. This is a TODO.

5.8.2 Alignment of frame may be critical

Let's go back to the example of the call center's timeboard with compound workshifts and weekend breaks. This is it:

```
>>> shifts_order = tb.RememberingPattern(['A', 'B', 'C', 'D'])
>>> day_parts = tb.Marker(each='D',
...                       at=[{'hours':2}, {'hours':8}, {'hours':18}])
>>> shifts = tb.Organizer(marker=day_parts, structure=shifts_order)
>>> week = tb.Marker(each='W',
...                  at=[{'days':0, 'hours':2}, {'days':5, 'hours':2}])
>>> weekly = tb.Organizer(marker=week, structure=[0, shifts])
>>> clnd = tb.Timeboard(base_unit_freq='H',
```

(continues on next page)

(continued from previous page)

```
...                     start='02 Oct 2017 00:00', end='10 Oct 2017 01:59',
...                     layout=weekly)
>>> clnd.add_schedule(name='team_A', selector=lambda label: label=='A')
>>>
>>> print(clnd)
Timeboard of 'H': 2017-10-02 00:00 -> 2017-10-10 01:00

                 ws_ref ... dur.                 end label  on_duty  team_A
loc                     ...
0   2017-10-02 00:00:00 ...    2 2017-10-02 01:59:59     0    False   False
1   2017-10-02 02:00:00 ...    6 2017-10-02 07:59:59     A     True    True
2   2017-10-02 08:00:00 ...   10 2017-10-02 17:59:59     B     True   False
3   2017-10-02 18:00:00 ...    8 2017-10-03 01:59:59     C     True   False
4   2017-10-03 02:00:00 ...    6 2017-10-03 07:59:59     D     True   False
5   2017-10-03 08:00:00 ...   10 2017-10-03 17:59:59     A     True    True
6   2017-10-03 18:00:00 ...    8 2017-10-04 01:59:59     B     True   False
7   2017-10-04 02:00:00 ...    6 2017-10-04 07:59:59     C     True   False
8   2017-10-04 08:00:00 ...   10 2017-10-04 17:59:59     D     True   False
9   2017-10-04 18:00:00 ...    8 2017-10-05 01:59:59     A     True    True
10  2017-10-05 02:00:00 ...    6 2017-10-05 07:59:59     B     True   False
11  2017-10-05 08:00:00 ...   10 2017-10-05 17:59:59     C     True   False
12  2017-10-05 18:00:00 ...    8 2017-10-06 01:59:59     D     True   False
13  2017-10-06 02:00:00 ...    6 2017-10-06 07:59:59     A     True    True
14  2017-10-06 08:00:00 ...   10 2017-10-06 17:59:59     B     True   False
15  2017-10-06 18:00:00 ...    8 2017-10-07 01:59:59     C     True   False
16  2017-10-07 02:00:00 ...   48 2017-10-09 01:59:59     0    False   False
17  2017-10-09 02:00:00 ...    6 2017-10-09 07:59:59     D     True   False
18  2017-10-09 08:00:00 ...   10 2017-10-09 17:59:59     A     True    True
19  2017-10-09 18:00:00 ...    8 2017-10-10 01:59:59     B     True   False

# The "start" column has been omitted and "duration" squeezed to fit
# the output to the page
```

However, if the start of the timeboard is moved to 02:00 of Monday or any time afterwards, the result will be totally incorrect:

```
>>> clnd = tb.Timeboard(base_unit_freq='H',
...                     start='02 Oct 2017 02:00', end='10 Oct 2017 01:59',
...                     layout=weekly)
>>> print(clnd)
Timeboard of 'H': 2017-10-02 02:00 -> 2017-10-10 01:00

                 ws_ref ... duration                 end label  on_duty
loc                     ...
0   2017-10-02 20:00:00 ...      102 2017-10-07 01:59:59     0    False
1   2017-10-07 02:00:00 ...        6 2017-10-07 07:59:59     C     True
2   2017-10-07 08:00:00 ...       10 2017-10-07 17:59:59     D     True
3   2017-10-07 18:00:00 ...        8 2017-10-08 01:59:59     A     True
4   2017-10-08 02:00:00 ...        6 2017-10-08 07:59:59     B     True
5   2017-10-08 08:00:00 ...       10 2017-10-08 17:59:59     C     True
6   2017-10-08 18:00:00 ...        8 2017-10-09 01:59:59     D     True
7   2017-10-09 02:00:00 ...       24 2017-10-10 01:59:59     0    False
```

What happened is the organizer having produced one span less than we expected when putting together the value of *structure*. We counted on the frame being aligned with a week. Thus the first element of structure, `0`, should have been applied to the span covering the period up to 01:59 of Monday. However, when the start of the frame moved to 02:00, the sequence of spans produced by the organizer was displaced in relation to the sequence of elements in *structure*.

Therefore, the elements of *structure* are now applied to inappropriate spans.

Workarounds:

- The most obvious solution is to swap elements of *structure*: `structure=[shifts, 0]`. However, this approach may render timeboard's configuration less comprehensible and more error-prone especially when elements of structure are related to specific days of the week or of months.
- The better approach is to align the start of the timeboard with boundaries of all calendar frequencies used in the timeboard's configuration.

  For example, if the base unit is an hour and `'W'` and `'D'` frequencies are used in organizers, start the timeboard at 00:00 Monday. If `'M'` frequency is used instead, start the timeboard at 00:00 of the first day of a month.

There is also another side effect to note. When we rebuilt the timeboard from 02:00 of Monday, you might have noticed that the pattern of labels in this *new* timeboard started on 'C', not on 'A'. This is because we continued to use the same layout that eventually references `RememberingPattern` *shifts_order* which has remembered where it stopped in the previous timeboard.

5.8.3 Specific days of month

A recurrent meeting gathers on the 10th, 20th and 30th day of the month.

The full-blown Marker-based approach is somewhat cumbersome and may produce obscure errors, like in this timeboard which breaks after April 30:

```
>>> days_of_month = tb.Marker(each='M',
...                           at=[{'days':9}, {'days':10}, {'days':19},
...                               {'days':20}, {'days':29}, {'days':30} ])
>>> monthly = tb.Organizer(marker=days_of_month,
...                        structure=[[0],[1],[0],[1],[0],[1]])
>>> clnd = tb.Timeboard(base_unit_freq='D',
...                     start='01 Jan 2017', end='31 Dec 2017',
...                     layout=monthly)
```

A straightforward technique facilitated by use of numpy's *zeros* is the best:

```
>>> import numpy as np
>>> days = np.zeros(31)
>>> days[[9,19,29]]=1
>>> monthly = tb.Organizer(marker='M',
...                        structure=[days])
>>> clnd = tb.Timeboard(base_unit_freq='D',
...                     start='01 Jan 2017', end='31 Dec 2017',
...                     layout=monthly)
```

CHAPTER 6

Using Preconfigured Calendars

There are a few preconfigured Timeboards that come with the package. They implement common business day calendars of different countries.

To access calendars of a country you have to import the country module from `timeboard.calendars`, for example:

```
>>> import timeboard.calendars.US as US
```

Then, to obtain a Timeboard implementing a required calendar, call the class for this calendar from the chosen module. Usually, the class takes some country-specific parameters that allow tuning the calendar. For example:

```
>>> clnd = US.Weekly8x5(do_not_observe = {'black_friday'})
```

`parameters()` class method returns the dictionary of the parameters used to instantiate the Timeboard. Of these, the most usable are probably parameters *start* and *end* which limit the maximum supported span of the calendar:

```
>>> params = US.Weekly8x5.parameters()
>>> params['start']
Timestamp('2000-01-01 00:00:00')
>>> params['end']
Timestamp('2020-12-31 23:59:59')
```

The currently available calendars are listed below. Consult the reference page of the calendar class to review its parameters and examples.

Country	Module	Calendar	Description
Russia	RU	`Weekly8x5`	Official calendar for 5 days x 8 hours working week with holiday observations
United Kingdom	UK	`Weekly8x5`	Business calendar for 5 days x 8 hours working week with bank holidays
United States	US	`Weekly8x5`	Business calendar for 5 days x 8 hours working week with federal holidays

CHAPTER 7

Doing Calculations

Table of Contents

Calendar calculations are performed either with an individual workshift or with an interval of workshifts.

Also, each calculation is based on a specific schedule in order to reason about duty statuses of workshifts involved.

Therefore, to carry out a calculation you need to obtain either a workshift or an interval and indicate which schedule you will be using.

The import statement to run the examples:

```
>>> import timeboard as tb
```

7.1 Obtaining a Workshift

Most likely you will want to identify a workshift by a timestamp which represents a point in time somewhere within the workshift. This is done by calling `Timeboard.get_workshift()` . The result returned will be an instance of `Workshift`.

```
>>> clnd = tb.Timeboard('D', '30 Sep 2017', '11 Oct 2017',
...                     layout=[0, 1, 0, 2])
>>> clnd.get_workshift('01 Oct 2017')
Workshift(1) of 'D' at 2017-10-01
```

Even simpler, you get the same result by calling the instance of `Timeboard` which will invoke `get_workshift()` for you:

```
>>> clnd('01 Oct 2017')
Workshift(1) of 'D' at 2017-10-01
```

The argument passed to `get_workshift()` is *Timestamp*-like meaning it may be a timestamp, or a string convertible to timestamp, or an object which implement *to_timestamp()* method.

Alternatively, you can call `Workshift()` constructor directly if you know the workshift's position on the timeline:

```
>>> tb.Workshift(clnd, 1)
Workshift(1) of 'D' at 2017-10-01
```

Every workshift comes with an attached schedule. This schedule is used in calculations carried out with this workshift unless it is overridden by *schedule* parameter of the method called to perform the calculation.

By default, a new workshift returned by `get_workshift()` method or `Workshift()` constructor receives the default schedule of the timeboard. You may attach a specific schedule to a new workshift by passing it in *schedule* parameter:

```
>>> sdl = clnd.add_schedule(name='my_schedule',
...                         selector=lambda label: label>1)
```

```
>>> clnd.get_workshift('01 Oct 2017', schedule=sdl)
Workshift(1, my_schedule) of 'D' at 2017-10-01
>>> tb.Workshift(clnd, 1, sdl)
Workshift(1, my_schedule) of 'D' at 2017-10-01
```

Note: You cannot obtain a workshift by calling the instance of `Timeboard` if you want to attach the schedule.‘ Use `get_workshift()` only.

Besides, a workshift can be obtained as a return value of a method performing a calculation over the timeboard. The schedule attached to this workshift is the schedule used by the method which has produced the workshift.

7.2 Workshift-based calculations

Method	Result
`is_on_duty()`	*Find out if the workshift is on duty.*
`is_off_duty()`	*Find out if the workshift is off duty.*
`worktime()`	*Return workshift's work time.*
`rollforward()`	*Return a workshift by taking the specified number of steps toward the future.*
+ (plus)	Shortcut for `rollforward()`
`rollback()`	*Return a workshift by taking the specified number of steps toward the past.*
- (minus)	Shortcut for `rollback()`

Each of the above methods must use some schedule to identify workshift's duty. The schedule is selected as follows:

- if a schedule is explicitly given as method's parameter, then use this schedule;
- else use the schedule attached to this workshift when it has been instantiated;
- if no *schedule* parameter was given to the workshift constructor, use the default schedule of the timeboard.

7.2.1 Determining duty

Examples:

```
>>> clnd = tb.Timeboard('D', '30 Sep 2017', '11 Oct 2017',
...                     layout=[0, 1, 0, 2])
>>> my_schedule = clnd.add_schedule(name='my_schedule',
                                    selector=lambda label: label>1)
```

```
>>> ws1 = clnd.get_workshift('01 Oct 2017')
>>> ws2 = clnd.get_workshift('01 Oct 2017', schedule=my_schedule)
```

ws1 and *ws2* are the same workshift but with different schedules attached. *ws1* comes with the default schedule of the timeboard, while *ws2* is given *my_schedule*.

The workshift has label `1`. Its duty under the default schedule:

```
>>> ws1.is_on_duty()
True
>>> ws2.is_on_duty(schedule=clnd.default_schedule)
True
```

and under *my_schedule*:

```
>>> ws1.is_on_duty(schedule=my_schedule)
False
>>> ws2.is_on_duty()
False
```

7.2.2 Obtaining work time

The source of the information about workshift's work time is determined by `Timeboard.worktime_source` attribute.

`Workshift.worktime()` method returns the work time of the workshift if the duty value passed to the method corresponds to that of the workshift. Otherwise, it returns zero.

By default, the work time equals to workshift's duration:

```
>>> clnd = tb.Timeboard('D', '30 Sep 2017', '11 Oct 2017',
...                     layout=[4, 8, 4, 8],
...                     default_selector = lambda label: label>4)
>>> ws = tb.Workshift(clnd, 3)
>>> ws.label
8.0
>>> ws.duration
1
>>> ws.is_on_duty()
True
>>> ws.worktime()
1
>>> ws.worktime(duty='off')
0
>>> ws.worktime(duty='any')
1
```

In the example below, the work time is taken from the labels:

```
>>> clnd = tb.Timeboard('D', '30 Sep 2017', '11 Oct 2017',
...                     layout=[4, 8, 4, 8],
...                     default_selector = lambda label: label>4,
...                     worktime_source = 'labels')
```

```
>>> ws = tb.Workshift(clnd, 3)
>>> ws.worktime()
8.0
>>> ws.worktime(duty='off')
0
>>> ws.worktime(duty='any')
8.0
```

```
>>> ws = tb.Workshift(clnd, 2)
>>> ws.label
4.0
>>> ws.is_off_duty()
True
>>> ws.worktime()
0
>>> ws.worktime(duty='off')
4.0
>>> ws.worktime(duty='any')
4.0
```

The query with `duty='off'` can be interpreted as "What is the work time for a worker who comes in when the main workforce is off duty?"

7.2.3 Rolling forward and back

The methods `rollforward()` and `rollback()` allow to identify the workshift which is located in a specified distance from the current workshift.

Actually, the methods do not roll, they step. The distance is measured in a number of steps with regard to a certain duty. It means that, when taking steps, the methods tread only on the workshifts with this duty, ignoring all others.

rollforward and *rollback* operate in the same manner except for the direction of time. You specify the number of steps and the duty to tread on. The default values are `steps=0, duty='on'`. The algorithm has two stages.

Stage 1. If you call a method omitting the number of steps (same as `steps=0`) it finds the closest workshift with the required duty.

```
>>> clnd = tb.Timeboard('D', '30 Sep 2017', '11 Oct 2017',
...                     layout=[0, 1, 0, 2])
>>> print(clnd)
Timeboard of 'D': 2017-09-30 -> 2017-10-11

        ws_ref      start  duration        end  label  on_duty
loc
0   2017-09-30 2017-09-30         1 2017-09-30    0.0    False
1   2017-10-01 2017-10-01         1 2017-10-01    1.0     True
2   2017-10-02 2017-10-02         1 2017-10-02    0.0    False
3   2017-10-03 2017-10-03         1 2017-10-03    2.0     True
4   2017-10-04 2017-10-04         1 2017-10-04    0.0    False
5   2017-10-05 2017-10-05         1 2017-10-05    1.0     True
6   2017-10-06 2017-10-06         1 2017-10-06    0.0    False
7   2017-10-07 2017-10-07         1 2017-10-07    2.0     True
8   2017-10-08 2017-10-08         1 2017-10-08    0.0    False
9   2017-10-09 2017-10-09         1 2017-10-09    1.0     True
10  2017-10-10 2017-10-10         1 2017-10-10    0.0    False
11  2017-10-11 2017-10-11         1 2017-10-11    2.0     True
```

```
>>> clnd('05 Oct 2017').rollforward()
Workshift(5) of 'D' at 2017-10-05
>>> clnd('06 Oct 2017').rollforward()
Workshift(7) of 'D' at 2017-10-07
```

```
>>> clnd('05 Oct 2017').rollback()
Workshift(5) of 'D' at 2017-10-05
>>> clnd('06 Oct 2017').rollback()
Workshift(5) of 'D' at 2017-10-05
```

A method returns the self workshift if its duty is the same as the duty sought. Otherwise it returns the next (*rollforward*) or the previous (*rollback*) workshift with the required duty. The example above illustrates this behavior for `duty='on'`, the example below - for `duty='off'`:

```
>>> clnd('05 Oct 2017').rollforward(duty='off')
Workshift(6) of 'D' at 2017-10-06
>>> clnd('06 Oct 2017').rollforward(duty='off')
Workshift(6) of 'D' at 2017-10-06
```

```
>>> clnd('05 Oct 2017').rollback(duty='off')
Workshift(4) of 'D' at 2017-10-04
>>> clnd('06 Oct 2017').rollback(duty='off')
Workshift(6) of 'D' at 2017-10-06
```

The result of stage 1 is called the "zero step workshift".

Stage 2. If the number of steps is not zero, a method proceeds to stage 2. After the zero step workshift has been found the method takes the required number of steps in the appropriate direction treading only on the workshifts with the specified duty:

```
>>> clnd('05 Oct 2017').rollforward(2)
Workshift(9) of 'D' at 2017-10-09
>>> clnd('06 Oct 2017').rollforward(2)
Workshift(11) of 'D' at 2017-10-11
```

```
>>> clnd('05 Oct 2017').rollback(2)
Workshift(1) of 'D' at 2017-10-01
>>> clnd('06 Oct 2017').rollback(2)
Workshift(1) of 'D' at 2017-10-01
```

```
>>> clnd('05 Oct 2017').rollforward(2, duty='off')
Workshift(10) of 'D' at 2017-10-10
>>> clnd('06 Oct 2017').rollforward(2, duty='off')
Workshift(10) of 'D' at 2017-10-10
```

```
>>> clnd('05 Oct 2017').rollback(2, duty='off')
Workshift(0) of 'D' at 2017-09-30
>>> clnd('06 Oct 2017').rollback(2, duty='off')
Workshift(2) of 'D' at 2017-10-02
```

Note: If you don't care about the duty and want to step on all workshifts, use `duty='any'`. This way the zero step workshift is always self.

As with the other methods, you can override the workshift's schedule in method's parameter. Take note that the returned workshift will have the schedule used by the method:

```
>>> my_schedule = clnd.add_schedule(name='my_schedule',
...                                 selector=lambda label: label>1)
>>> ws = clnd('05 Oct 2017').rollforward(schedule=my_schedule)
>>> ws
Workshift(7, my_schedule) of 'D' at 2017-10-07
>>> ws.rollforward(1)
Workshift(11, my_schedule) of 'D' at 2017-10-11
```

Using operators + and -

You can add or subtract an integer number to/from a workshift. This is the same as calling, accordingly, *rollforward* or *rollback* with `duty='on'`.

```
# under default schedule
>>> clnd('05 Oct 2017') + 1
Workshift(7) of 'D' at 2017-10-07
>>> clnd('06 Oct 2017') - 1
Workshift(3) of 'D' at 2017-10-03
```

```
# under my_schedule
>>> ws = clnd.get_workshift('05 Oct 2017', schedule=my_schedule)
>>> ws + 1
Workshift(11, my_schedule) of 'D' at 2017-10-11
```

Caveats

steps can take a negative value. A method will step in the opposite direction, however, the algorithm of seeking the zero step workshift does not change. Therefore, the results of *rollforward* with negative *steps* and *rollback* with the same but positive value of *steps* may differ:

```
>>> clnd('06 Oct 2017').rollforward(-1)
Workshift(5) of 'D' at 2017-10-05
>>> clnd('06 Oct 2017').rollback(1)
Workshift(3) of 'D' at 2017-10-03
```

As the workshift of October 6 is off duty while method's duty is "on" by default, the method must seek the zero step workshift. In doing that, *rollforward* looks in the future and finds October 7, while *rollback* looks in the past and find October 5. Then both methods take one "on-duty" step to the past and arrive at the results shown above.

The analogous behavior takes place with `rollback(-n)` and `rollforward(n)`:

```
>>> clnd('05 Oct 2017').rollback(-1, duty='off')
Workshift(6) of 'D' at 2017-10-06
>>> clnd('05 Oct 2017').rollforward(1, duty='off')
Workshift(8) of 'D' at 2017-10-08
```

There is no such discrepancy if method's duty is the same as workshift's duty.

7.3 Obtaining an Interval

Method	Result
`Timeboard.get_interval()`	Create an interval with regard to specific points or periods of time: from two points in time, or from a calendar period, or specify the starting point and the length of the interval.
calling *Timeboard* instance	Shortcut for `Timeboard.get_interval()`
`Interval()`	Instantiate an interval from the first and the last workshifts or from their sequence numbers on the timeline.
`Interval.overlap()`	Get an interval that is the intersection of two intervals.
* (multiplication)	Shortcut for `overlap()`

To create an interval with regard to the specific points or periods of time call `Timeboard.get_interval()`. This method takes several combinations of parameters. In most cases, you can also use a shortcut by calling the instance of `Timeboard` which will invoke `get_interval()` for you.

Obtaining an interval from two points in time:

```
>>> clnd = tb.Timeboard('D', '30 Sep 2017', '15 Oct 2017',
...                     layout=[0, 1, 0, 2])
>>> clnd.get_interval(('02 Oct 2017', '08 Oct 2017'))
Interval((2, 8)): 'D' at 2017-10-02 -> 'D' at 2017-10-08 [7]

# Shortcut:

>>> clnd(('02 Oct 2017', '08 Oct 2017'))
Interval((2, 8)): 'D' at 2017-10-02 -> 'D' at 2017-10-08 [7]
```

The points in time come as a tuple of two values which are timestamps, or strings convertible to timestamps, or objects which implement *to_timestamp()* method.

Note that the points in time are not the boundaries of the interval but references to the first and the last workshifts of the interval. The points in time may be located anywhere within these workshifts. The following operation produces the same interval as the one above:

```
>>> clnd.get_interval(('02 Oct 2017 15:15', '08 Oct 2017 23:59'))
Interval((2, 8)): 'D' at 2017-10-02 -> 'D' at 2017-10-08 [7]
```

You may also pass a null value (such as *None*, *NaN*, or *NaT*) in place of a point in time. If the first element of the tuple is null, then the interval will start on the first workshift of the timeboard. If the second element is null, then the interval will end on the last workshift of the timeboard.

```
>>> clnd.get_interval((None, '08 Oct 2017 23:59'))
Interval((0, 8)): 'D' at 2017-09-30 -> 'D' at 2017-10-08 [9]
>>> clnd(('02 Oct 2017 15:15', None))
Interval((2, 15)): 'D' at 2017-10-02 -> 'D' at 2017-10-15 [14]
```

Building an interval of a specified length:

```
>>> clnd.get_interval('02 Oct 2017', length=7)
Interval((2, 8)): 'D' at 2017-10-02 -> 'D' at 2017-10-08 [7]

# Shortcut:

>>> clnd('02 Oct 2017', length=7)
Interval((2, 8)): 'D' at 2017-10-02 -> 'D' at 2017-10-08 [7]
```

Obtaining an interval from a calendar period:

```
>>> clnd.get_interval('05 Oct 2017', period='W')
Interval((2, 8)): 'D' at 2017-10-02 -> 'D' at 2017-10-08 [7]

# Shortcut:

>>> clnd('05 Oct 2017', period='W')
Interval((2, 8)): 'D' at 2017-10-02 -> 'D' at 2017-10-08 [7]
```

You can also build an interval directly from *pandas.Period* object but the shortcut is not available:

```
>>> import pandas as pd
>>> p = pd.Period('05 Oct 2017', freq='W')
>>> clnd.get_interval(p)
Interval((2, 8)): 'D' at 2017-10-02 -> 'D' at 2017-10-08 [7]

# NO shortcut!

>>> clnd(p)
Workshift(2) of 'D' at 2017-10-02
```

Finally, you can convert the entire timeline into the interval:

```
>>> clnd.get_interval()
Interval((0, 15)): 'D' at 2017-09-30 -> 'D' at 2017-10-15 [16]

# Shortcut:
```

(continues on next page)

(continued from previous page)

```
>>> clnd()
Interval((0, 15)): 'D' at 2017-09-30 -> 'D' at 2017-10-15 [16]
```

Alternatively, you can call `Interval()` constructor directly if you have got the first and the last workshifts of the interval or know their sequence numbers on the timeline:

```
>>> ws_first = clnd('02 Oct 2017')
>>> ws_first
Workshift(2) of 'D' at 2017-10-02
>>> ws_last = clnd('08 Oct 2017')
>>> ws_last
Workshift(8) of 'D' at 2017-10-08
```

```
>>> tb.Interval(clnd, (ws_first, ws_last))
Interval((2, 8)): 'D' at 2017-10-02 -> 'D' at 2017-10-08 [7]
```

```
>>> tb.Interval(clnd, (2, 8))
Interval((2, 8)): 'D' at 2017-10-02 -> 'D' at 2017-10-08 [7]
```

If you have got two intervals you can obtain an interval representing their intersection by calling `overlap()` on any of the two while passing the other as the parameter:

```
>>> ivl = tb.Interval(clnd, (2, 8))
>>> other = tb.Interval(clnd, (6, 10))
```

```
>>> ivl.overlap(other)
Interval((6, 8)): 'D' at 2017-10-06 -> 'D' at 2017-10-08 [3]
```

As a shortcut, * (multiplication) operator can be used:

```
>>> ivl * other
Interval((6, 8)): 'D' at 2017-10-06 -> 'D' at 2017-10-08 [3]
```

Every interval comes with an attached schedule. This schedule is used in calculations carried out with this interval unless it is overridden by *schedule* parameter of the method called to perform the calculation.

By default, a new interval receives the default schedule of the timeboard or inherits the schedule from its parent interval (i.e. from the interval on which *overlap()* has been called).

You may attach a specific schedule to a new interval by passing it in *schedule* parameter of any method you use to instantiate an interval:

```
>>> my_schedule = clnd.add_schedule(name='my_schedule',
...                                 selector=lambda label: label>1)
```

```
>>> clnd(('02 Oct 2017', '08 Oct 2017'), schedule=my_schedule)
Interval((2, 8), my_schedule): 'D' at 2017-10-02 -> 'D' at 2017-10-08 [7]
>>> tb.Interval(clnd, (2,8), schedule=my_schedule)
Interval((2, 8), my_schedule): 'D' at 2017-10-02 -> 'D' at 2017-10-08 [7]
>>> ivl.overlap(other, schedule=my_schedule)
Interval((6, 8), my_schedule): 'D' at 2017-10-06 -> 'D' at 2017-10-08 [3]
```

7.3.1 Caveats

There are a few caveats when you instantiate an interval from a calendar period.

Period extends beyond timeline

If the calendar period extends beyond the timeline, the interval is created as the intersection of the timeline and the calendar period.

```
>>> clnd = tb.Timeboard('D', '30 Sep 2017', '15 Oct 2017',
...                     layout=[0, 1, 0, 2])
>>> clnd('Oct 2017', period='M')
Interval(1, 15): 'D' at 2017-10-01 -> 'D' at 2017-10-15 [15]
```

There is a parameter called *clip_period* which determines how this situation is handled. By default `clip_period=True` which results in the behavior illustrated above. If it is set to False, *PartialOutOfBoundsError* is raised:

```
>>> clnd('Oct 2017', period='M', clip_period=False)
---------------------------------------------------------------------------
PartialOutOfBoundsError                Traceback (most recent call last)
  ...
PartialOutOfBoundsError: The right bound of interval referenced by `Oct
2017` is outside Timeboard of 'D': 2017-09-30 -> 2017-10-15
```

Workshift straddles period boundary

Consider the following timeboard:

```
>>> clnd = tb.Timeboard('12H', '01 Oct 2017 21:00', '03 Oct 2017',
...                     layout=[1])
>>> print(clnd)

                 ws_ref               start  duration                 end
loc
0   2017-10-01 21:00:00 2017-10-01 21:00:00         1 2017-10-02 08:59:59
1   2017-10-02 09:00:00 2017-10-02 09:00:00         1 2017-10-02 20:59:59
2   2017-10-02 21:00:00 2017-10-02 21:00:00         1 2017-10-03 08:59:59

# columns "label" and "on_duty" have been omitted to fit the output
# to the page
```

Suppose we want to build an interval corresponding to the day of October 2. The workshifts at locations 0 and 2 straddle the boundaries of the day: they partly lay within October 2 and partly - without.

This ambiguity is solved with `Timeboard.workshift_ref` attribute. The workshift is considered a member of the calendar period where its reference timestamp belongs. By default, workshift's reference timestamp is its start time (`workshift_ref='start'`). This is shown in column 'workshift' in the output above. Hence, workshift's membership in a calendar period is determined by its start time. In our timeboard, consequently, workshift 0 belongs to October 1 while workshift 2 stays with October 2:

```
>>> clnd('02 Oct 2017', period='D')
Interval((1, 2)): '12H' at 2017-10-02 09:00 -> '12H' at 2017-10-02 21:00 [2]
```

Note the change in 'workshift' column in the output below when `workshift_ref='end'`:

```
>>> clnd = tb.Timeboard('12H', '01 Oct 2017 21:00', '03 Oct 2017',
...                     layout=[1],
...                        ws_ref_ref='end')
>>> print(clnd)
```

(continues on next page)

(continued from previous page)

```
Timeboard of '12H': 2017-10-01 21:00 -> 2017-10-02 21:00

                 ws_ref               start  duration                 end
loc
0   2017-10-02 08:59:59 2017-10-01 21:00:00         1 2017-10-02 08:59:59
1   2017-10-02 20:59:59 2017-10-02 09:00:00         1 2017-10-02 20:59:59
2   2017-10-03 08:59:59 2017-10-02 21:00:00         1 2017-10-03 08:59:59

# columns "label" and "on_duty" have been omitted to fit the output
# to the page
```

In this way, the end time of workshift is used as the indicator of period membership. Workshift 0 becomes a member of October 2 while workshift 2 goes with October 3:

```
>>> clnd('02 Oct 2017', period='D')
Interval((0, 1)): '12H' at 2017-10-01 21:00 -> '12H' at 2017-10-02 09:00 [2]
```

Due to the skewed workshift alignment, in both cases the boundaries of the produced interval do not coincide with the period given as the interval reference (the day of October 2).

Period too short for workshifts

In a corner case, you can try to obtain an interval from a period which is shorter than the workshifts in this area of the timeline. For example, in a timeboard with daily workshifts you seek an interval defined by an hour:

```
>>> clnd = tb.Timeboard('D', '30 Sep 2017', '05 Oct 2017', layout=[1])
>>> ivl = clnd.get_interval('02 Oct 2017 00:00', period='H')
```

However meaningless, this operation is handled according to the same logic of attributing a workshift to the period as discussed in the previous section. In this timeboard, the workshift reference time is its start time (the default setting). The hour starting at 02 Oct 2017 00:00 contains the reference time of the daily workshift of October 2. Technically, this one-day workshift is the member of the one-hour period and, therefore, becomes the only element of the sought interval:

```
>>> print(ivl)
Interval((2, 2)): 'D' at 2017-10-02 -> 'D' at 2017-10-02 [1]

        ws_ref      start  duration        end  label  on_duty
loc
2   2017-10-02 2017-10-02         1 2017-10-02    1.0     True
```

On the other hand, if you try to obtain an interval from another hour of the same day, *VoidIntervalError* will be raised as no workshift has its reference time within that hour:

```
>>> clnd.get_interval('02 Oct 2017 01:00', period='H')
---------------------------------------------------------------------------
VoidIntervalError                         Traceback (most recent call last)
  ...
VoidIntervalError: Attempted to create reversed or void interval
referenced by `02 Oct 2017 01:00` within Timeboard of 'D': 2017-09-30 ->
2017-10-05
```

7.4 Interval-based calculations

Method	Result
`nth()`	*Find n-th workshift with the specified duty in the interval.*
`first()`	*Find the first workshift with the specified duty in the interval.*
`last()`	*Find the last workshift with the specified duty in the interval.*
`workshifts()`	*Iterate through workshifts with the specified duty.*
`count()`	*Count workshifts with the specified duty in the interval.*
`worktime()`	*The total work time of workshifts with the specified duty.*
`what_portion_of()`	*What portion of another interval this interval takes up.*
`/` (division)	Shortcut for `what_portion_of()`
`count_periods()`	*How many calendar periods fit into the interval.*

All methods are duty-aware meaning that they "see" only workshifts with the specified duty ignoring the others.

Each of the above methods must use some schedule to identify workshift's duty. The schedule is selected as follows:

- if a schedule is explicitly given as method's parameter, then use this schedule;
- else use the schedule attached to this interval when it has been instantiated;
- if no *schedule* parameter was given to the interval constructor, use the default schedule of the timeboard.

Note: If you don't care about the duty and want to take into account all workshifts in the interval, use `duty='any'`.

7.4.1 Seeking and counting workshifts

Create an interval for the examples:

```
>>> clnd = tb.Timeboard('D', '30 Sep 2017', '15 Oct 2017',
...                     layout=[0, 1, 0, 2])
>>> ivl = clnd(('02 Oct 2017', '08 Oct 2017'))
>>> print(ivl)
Interval((2, 8)): 'D' at 2017-10-02 -> 'D' at 2017-10-08 [7]

        ws_ref      start  duration        end  label  on_duty
loc
2   2017-10-02 2017-10-02         1 2017-10-02    0.0    False
3   2017-10-03 2017-10-03         1 2017-10-03    2.0     True
4   2017-10-04 2017-10-04         1 2017-10-04    0.0    False
5   2017-10-05 2017-10-05         1 2017-10-05    1.0     True
6   2017-10-06 2017-10-06         1 2017-10-06    0.0    False
7   2017-10-07 2017-10-07         1 2017-10-07    2.0     True
8   2017-10-08 2017-10-08         1 2017-10-08    0.0    False
```

Seeking and counting with `duty='on'`:

```
>>> ivl.first()
Workshift(3) of 'D' at 2017-10-03
>>> ivl.nth(1)
Workshift(5) of 'D' at 2017-10-05
>>> ivl.last()
Workshift(7) of 'D' at 2017-10-07
```

(continues on next page)

(continued from previous page)

```
>>> ivl.count()
3
```

With `duty='off'`:

```
>>> ivl.first(duty='off')
Workshift(2) of 'D' at 2017-10-02
>>> ivl.nth(1, duty='off')
Workshift(4) of 'D' at 2017-10-04
>>> ivl.last(duty='off')
Workshift(8) of 'D' at 2017-10-08
>>> ivl.count(duty='off')
4
```

With `duty='on'` under another schedule:

```
>>> my_schedule = clnd.add_schedule(name='my_schedule',
...                                 selector=lambda label: label>1)
>>> ivl.nth(1, schedule=my_schedule)
Workshift(7, my_schedule) of 'D' at 2017-10-07
>>> ivl.count(duty='on', schedule=my_schedule)
2
```

Not taking the duty into account:

```
>>> ivl.first(duty='any')
Workshift(2) of 'D' at 2017-10-02
>>> ivl.nth(1, duty='any')
Workshift(3) of 'D' at 2017-10-03
>>> ivl.last(duty='any')
Workshift(8) of 'D' at 2017-10-08
>>> ivl.count(duty='any')
7
```

7.4.2 Itertating over the interval

`workshifts()` returns a generator that iterates over the interval and yields workshifts with the specified duty. By default, the duty is "on".

```
>>> clnd = tb.Timeboard('D', '30 Sep 2017', '15 Oct 2017',
...                     layout=[0, 1, 0, 2])
>>> ivl = clnd(('02 Oct 2017', '08 Oct 2017'))
>>> print(ivl)
Interval((2, 8)): 'D' at 2017-10-02 -> 'D' at 2017-10-08 [7]

        ws_ref      start  duration        end  label  on_duty
loc
2   2017-10-02 2017-10-02         1 2017-10-02    0.0    False
3   2017-10-03 2017-10-03         1 2017-10-03    2.0     True
4   2017-10-04 2017-10-04         1 2017-10-04    0.0    False
5   2017-10-05 2017-10-05         1 2017-10-05    1.0     True
6   2017-10-06 2017-10-06         1 2017-10-06    0.0    False
7   2017-10-07 2017-10-07         1 2017-10-07    2.0     True
8   2017-10-08 2017-10-08         1 2017-10-08    0.0    False
```

```
>>> for ws in ivl.workshifts():
...     print("{}\t{}".format(ws.start_time, ws.label))
2017-10-03 00:00:00     2
2017-10-05 00:00:00     1
2017-10-07 00:00:00     2
```

```
>>> list(ivl.workshifts(duty='off'))
[Workshift(2) of 'D' at 2017-10-02,
 Workshift(4) of 'D' at 2017-10-04,
 Workshift(6) of 'D' at 2017-10-06,
 Workshift(8) of 'D' at 2017-10-08]
```

You can also use the interval itself as a generator that yields every workshift of the interval. This is the same generator as returned by `ivl.workshifts(duty='any')`.

```
>>> for ws in ivl:
...     print("{}\t{}".format(ws.start_time, ws.label))
2017-10-02 00:00:00 0
2017-10-03 00:00:00 2
2017-10-04 00:00:00 0
2017-10-05 00:00:00 1
2017-10-06 00:00:00 0
2017-10-07 00:00:00 2
2017-10-08 00:00:00 0
```

```
>>> list(ivl.workshifts(duty='any'))
[Workshift(2) of 'D' at 2017-10-02,
 Workshift(3) of 'D' at 2017-10-03,
 Workshift(4) of 'D' at 2017-10-04,
 Workshift(5) of 'D' at 2017-10-05,
 Workshift(6) of 'D' at 2017-10-06,
 Workshift(7) of 'D' at 2017-10-07,
 Workshift(8) of 'D' at 2017-10-08]
```

7.4.3 Measuring work time

The source of the information about workshifts' work time is determined by `Timeboard.worktime_source` attribute.

`Interval.worktime()` method returns the sum of the work times of the workshifts with the specified duty. If the interval does not contain workshifts with this duty, the method returns zero.

By default, workshift's work time equals to workshift's duration:

```
>>> clnd = tb.Timeboard('D', '30 Sep 2017', '11 Oct 2017',
...                     layout=[4, 8, 4, 8],
...                     default_selector = lambda label: label>4)
>>> ivl = tb.Interval(clnd, (1, 3))
>>> print (ivl)
Interval((1, 3)): 'D' at 2017-10-01 -> 'D' at 2017-10-03 [3]

        ws_ref      start  duration        end  label  on_duty
loc
1   2017-10-01 2017-10-01         1 2017-10-01    8.0     True
2   2017-10-02 2017-10-02         1 2017-10-02    4.0    False
3   2017-10-03 2017-10-03         1 2017-10-03    8.0     True
```

```
>>> ivl.worktime()
2
>>> ivl.worktime(duty='off')
1
>>> ivl.worktime(duty='any')
3
```

In the example below, the work time is taken from the labels:

```
>>> clnd = tb.Timeboard('D', '30 Sep 2017', '11 Oct 2017',
...                     layout=[4, 8, 4, 8],
...                     default_selector = lambda label: label>4,
...                     worktime_source = 'labels')
>>> ivl = tb.Interval(clnd, (1, 3))
```

```
>>> ivl.worktime()
16.0
>>> ivl.worktime(duty='off')
4.0
>>> ivl.worktime(duty='any')
20.0
```

Note: To count the total duration of the workshifts in the interval (regardless of the work time) call `Interval.total_duration()`.

7.4.4 Relation with another interval

`what_portion_of()` builds the intersection of this interval and another and returns the ratio of the workshift count in the intersection to the workshift count in the other interval. Only workshifts with the specified duty are counted.

If the two intervals do not overlap or their intersection contains no workshifts with the specified duty, zero is returned.

The common use of this method is to answer questions like "what portion of year 2017 has employee X been with the company?". In the examples below, for the purpose of demonstration, the question is scaled down to "what portion of the week?..":

```
>>> clnd = tb.Timeboard('D', '02 Oct 2017', '15 Oct 2017',
...                     layout=[1, 1, 1, 1, 1, 0, 0])
>>> week1 = clnd('02 Oct 2017', period='W')
```

week1 contains five working days and two days off.

```
>>> X_in_staff = clnd(('05 Oct 2017', '07 Oct 2017'))
```

X was was with the company Thursday through Saturday of *week1* (two working days and one day off).

```
>>> .what_portion_of(week1)
0.4
>>> 2 / 5 # working days
0.4
```

```
>>> X_in_staff.what_portion_of(week1, duty='off')
0.5
```

(continues on next page)

(continued from previous page)

```
>>> 1 / 2 # days off
0.5
```

```
>>> X_in_staff.what_portion_of(week1, duty='any')
0.42857142857142855
>>> 3 / 7 # all days
0.42857142857142855
```

You can use / (division) operator as a shortcut. It calls *what_portion_of()* with the default parameter values (so, the duty is 'on'):

```
>>> X_in_staff / week1
0.4
```

X had already left before *week2* started:

```
>>> week2 = clnd('09 Oct 2017', period='W')
>>> X_in_staff.what_portion_of(week2, duty='any')
0.0
```

Y has worked the entire *week1* and stayed afterwards:

```
>>> Y_in_staff = clnd(('02 Oct 2017', '11 Oct 2017'))
>>> decade.what_portion_of(week1)
1.0
```

A corner case:

```
>>> weekend = clnd(('07 Oct 2017', '08 Oct 2017'))
```

All days of *weekend* are also the days of *week1* but they are not working days, so:

```
>>> weekend.what_portion_of(week1)
0.0
```

However, *weekend* contains all off-duty days of *week1*:

```
>>> weekend.what_portion_of(week1, duty='off')
1.0
```

7.4.5 Counting periods

Call `count_periods()` to find out how many calendar periods of the specific frequency fit into the interval. As with the other methods, the duty of workshifts is taken into account. The method returns a float number.

To obtain the result, the interval is sliced into calendar periods of the given frequency and then each slice of the interval is compared to its corresponding period duty-wise. That is to say, the count of workshifts in the interval's slice is divided by the total count of workshifts in the period containing this slice but only workshifts with the specified duty are counted. The quotients for each period are summed to produce the return value of the method.

If some period does not contain workshifts of the required duty, it contributes zero to the returned value.

Regardless of the period frequency, the method returns 0.0 if there are no workshifts with the specified duty in the interval.

The common use of this method is to answer questions like "Exactly, how many years has X worked in the company?" In the examples below, for the purpose of demonstration, the question is scaled down to "how many days?.." for a timeboard with hourly shifts.

Examples:

```
>>> clnd = tb.Timeboard('H', '01 Oct 2017', '08 Oct 2017 23:59',
...                     layout=[0, 1, 0, 2])
>>> X_in_staff = clnd(('01 Oct 2017 13:00', '02 Oct 2017 23:59'))
```

X's tenure spans two days: it contains 11 of 24 workshifts of October 1, and all 24 workshifts of October 2:

```
>>> X_in_staff.count_periods('D', duty='any')
1.4583333333333333
>>> 11.0/24 + 24.0/24
1.4583333333333333
```

The timeboard's *layout* defines that all workshifts taking place on even hours are off duty, and those on odd hours are on duty. The first workshift of the interval (01 October 13:00 - 13:59) is on duty. Hence, interval *X_in_staff* contains 6 of 12 on-duty workshifts of October 1, and all 12 on-duty workshifts of October 2:

```
>>> X_in_staff.count_periods('D')
1.5
>>> 6.0/12 + 12.0/12
1.5
```

The interval contains 5 of 12 off-duty workshifts of October 1, and all 12 off-duty workshifts of October 2:

```
>>> X_in_staff.count_periods('D', duty='off')
1.4166666666666667
>>> 5.0/12 + 12.0/12
1.4166666666666667
```

If we change the schedule to *my_schedule*, on-duty workshifts will start only at 3, 7, 11, 15, 19, and 23 o'clock yielding 6 on-duty workshifts per day. Interval *X_in_staff* will contain 3/6 + 6/6 on-duty days and 8/18 + 18/18 off-duty days:

```
>>> my_schedule = clnd.add_schedule(name='my_schedule',
...                                 selector=lambda label: label>1)
```

```
>>> X_in_staff.count_periods('D', schedule=my_schedule)
1.5
>>> 3.0/6 + 6.0/6
1.5
>>> X_in_staff.count_periods('D', duty='off', schedule=my_schedule)
1.4444444444444444
>>> 8.0/18 + 18.0/18
1.4444444444444444
```

Note that an interval containing exactly one calendar period with regard to some duty may be larger than this period, as well as smaller:

```
# Interval of 25 hours
>>> ivl = clnd(('01 Oct 2017 00:00', '02 Oct 2017 00:59'))
>>> ivl
Interval((0, 24)): 'H' at 2017-10-01 00:00 -> 'H' at 2017-10-02 00:00 [25]
>>> ivl.count_periods('D')
1.0
```

```
# Interval of 23 hours
>>> ivl = clnd(('01 Oct 2017 01:00', '01 Oct 2017 23:59'))
>>> ivl
Interval((1, 23)): 'H' at 2017-10-01 01:00 -> 'H' at 2017-10-01 23:00 [23]
>>> ivl.count_periods('D')
1.0
```

7.4.6 Caveats

Period extends beyond timeline

Consider the timeboard and two intervals:

```
>>> clnd = tb.Timeboard('H', '01 Oct 2017', '08 Oct 2017 23:59',
...                     layout=[0, 1, 0, 2])
>>> ivl1 = clnd(('02 Oct 2017 00:00', '02 Oct 2017 23:59'))
>>> ivl2 = clnd(('01 Oct 2017 13:00', '02 Oct 2017 23:59'))
```

We can count how many weeks are in interval *ivl1* but not in *ivl2*.

All workshifts of *ivl1* belong to the week of October 2 - 8 which is situated entirely within the timeboard. On the other hand, in *ivl2* there are the workshifts belonging to the week of September 25 - October 1. This week extends beyond the timeboard. We may not guess what layout *could* be applied to the workshifts of Sep 25 - Sep 30 if the week were included in the timeboard entirely. We are not authorized to extrapolate the existing layout outside the timeboard. Moreover, for some complex layouts, any attempt at extrapolation would be ambiguous.

```
>>> ivl1.count_periods('W')
0.14285714285714285
>>> ivl2.count_periods('W')
---------------------------------------------------------------------------
PartialOutOfBoundsError                Traceback (most recent call last)
  ...
PartialOutOfBoundsError: The left bound of interval or period referenced by `2017-09-
↪25/2017-10-01` is outside Timeboard of 'H': 2017-10-01 00:00 -> 2017-10-08 23:00
```

Workshift straddles period boundary

This case is analogous to the already reviewed *issue* of constructing an interval from a calendar period. `Timeboard.workshift_ref` attribute is used to identify workshift's membership in a period.

Period too short for workshifts

If you try to count periods which are shorter than (some) of the workshifts in the interval, you are likely to encounter a period which does not contain *any* workshift's reference whatever the duty. This makes any result meaningless and, consequently, *UnacceptablePeriodError* is raised.

You may accidentally run into this issue in two situations:

- You use compound workshifts and while most of the workshifts (usually those covering the working time) are of one size, there are a few workshifts (usually those covering the closed time) which are much larger. Trying to count periods, you have in mind the smaller workshifts. If a larger one gets into the interval and your period is not long enough, you will find yourself with *UnacceptablePeriodError*.

- You have misinterpreted the purpose of `count_periods()` method and try to use it as a general time counter. For example, in a timeboard with workshifts of varying duration measured in hours, you want to find out how many clock hours there are in an interval. In order to do that use *pandas.Timedelta* tools with *start_time* and *end_time* attributes of workshifts and intervals.

CHAPTER 8

Common Use Cases

Table of Contents

This document contains code snippets for the common use cases of `timeboard` library. It is also available as a jupyter notebook.

The import statements for all examples are:

[1]:
```
import timeboard as tb
import pandas as pd
```

Note: We will use `pandas` dataframes to store the data we work with.

8.1 Setting up the calendar

Two types of calendars are used in the examples: a standard business day calendar and a timeboard of shifts in a 24x7 call center. The detailed explanations how to create these or other timeboards are given in *Making a Timeboard*

section. Calculations are performed similarly for any type of timeboard.

To obtain a standard business day calendar we use the built-ins:

[2]:
```
import timeboard.calendars.RU as RU
clnd_ru = RU.Weekly8x5()

import timeboard.calendars.UK as UK
clnd_uk = UK.Weekly8x5(country='england')
```

A sample of the UK calendar `clnd_uk` is shown below. It starts on Monday, the 17th of April, which was a holiday (Easter Monday), and ends on Monday the 24th, a regular business day.

Note: We take advantage of the nice formatting that jupyter notebooks provide for pandas dataframes. Instead of official `print(clnd_uk(('17 Apr 2017', '24 Apr 2017')))`, we will convert the interval to dataframe and let jupyter display its contents.

[3]:
```
clnd_uk(('17 Apr 2017', '24 Apr 2017')).to_dataframe()
```

[3]:

loc	ws_ref	start	duration	end	label	on_duty
6316	2017-04-17	2017-04-17	1	2017-04-17	0	False
6317	2017-04-18	2017-04-18	1	2017-04-18	8	True
6318	2017-04-19	2017-04-19	1	2017-04-19	8	True
6319	2017-04-20	2017-04-20	1	2017-04-20	8	True
6320	2017-04-21	2017-04-21	1	2017-04-21	8	True
6321	2017-04-22	2017-04-22	1	2017-04-22	0	False
6322	2017-04-23	2017-04-23	1	2017-04-23	0	False
6323	2017-04-24	2017-04-24	1	2017-04-24	8	True

Our call center operates round-the-clock in shifts of varying length: 08:00 to 18:00 (10 hours), 18:00 to 02:00 (8 hours), and 02:00 to 08:00 (6 hours). An operator's schedule consists of one on-duty shift followed by three off-duty shifts. Hence, four teams of operators are needed. They are designated as 'A', 'B', 'C', and 'D'. Timeboard `clnd_cc` for the call center is built by the following code.

[4]:
```
teams = ['A', 'B', 'C', 'D']
day_parts = tb.Marker(each='D',
                      at=[{'hours':2}, {'hours':8}, {'hours':18}])
shifts = tb.Organizer(marker=day_parts, structure=teams)
clnd_cc = tb.Timeboard(base_unit_freq='H',
                       start='01 Jan 2009 02:00', end='01 Jan 2019 01:59',
                       layout=shifts)
for team in teams:
    clnd_cc.add_schedule(name='team_'+ team,
                         selector=lambda label, team=team: label==team)
```

A sample of `clnd_cc` for the week of 17 April 2017 is shown below.

[5]:
```
clnd_cc(('17 Apr 2017 2:00', '24 Apr 2017')).to_dataframe()
```

[5]:

loc	ws_ref	start	duration	end	\
9084	2017-04-17 02:00:00	2017-04-17 02:00:00	6	2017-04-17 07:59:59	
9085	2017-04-17 08:00:00	2017-04-17 08:00:00	10	2017-04-17 17:59:59	
9086	2017-04-17 18:00:00	2017-04-17 18:00:00	8	2017-04-18 01:59:59	
9087	2017-04-18 02:00:00	2017-04-18 02:00:00	6	2017-04-18 07:59:59	
9088	2017-04-18 08:00:00	2017-04-18 08:00:00	10	2017-04-18 17:59:59	
9089	2017-04-18 18:00:00	2017-04-18 18:00:00	8	2017-04-19 01:59:59	
9090	2017-04-19 02:00:00	2017-04-19 02:00:00	6	2017-04-19 07:59:59	

(continues on next page)

(continued from previous page)

```
9091 2017-04-19 08:00:00 2017-04-19 08:00:00        10 2017-04-19 17:59:59
9092 2017-04-19 18:00:00 2017-04-19 18:00:00         8 2017-04-20 01:59:59
9093 2017-04-20 02:00:00 2017-04-20 02:00:00         6 2017-04-20 07:59:59
9094 2017-04-20 08:00:00 2017-04-20 08:00:00        10 2017-04-20 17:59:59
9095 2017-04-20 18:00:00 2017-04-20 18:00:00         8 2017-04-21 01:59:59
9096 2017-04-21 02:00:00 2017-04-21 02:00:00         6 2017-04-21 07:59:59
9097 2017-04-21 08:00:00 2017-04-21 08:00:00        10 2017-04-21 17:59:59
9098 2017-04-21 18:00:00 2017-04-21 18:00:00         8 2017-04-22 01:59:59
9099 2017-04-22 02:00:00 2017-04-22 02:00:00         6 2017-04-22 07:59:59
9100 2017-04-22 08:00:00 2017-04-22 08:00:00        10 2017-04-22 17:59:59
9101 2017-04-22 18:00:00 2017-04-22 18:00:00         8 2017-04-23 01:59:59
9102 2017-04-23 02:00:00 2017-04-23 02:00:00         6 2017-04-23 07:59:59
9103 2017-04-23 08:00:00 2017-04-23 08:00:00        10 2017-04-23 17:59:59
9104 2017-04-23 18:00:00 2017-04-23 18:00:00         8 2017-04-24 01:59:59

     label  on_duty  team_A  team_B  team_C  team_D
loc
9084     A     True    True   False   False   False
9085     B     True   False    True   False   False
9086     C     True   False   False    True   False
9087     D     True   False   False   False    True
9088     A     True    True   False   False   False
9089     B     True   False    True   False   False
9090     C     True   False   False    True   False
9091     D     True   False   False   False    True
9092     A     True    True   False   False   False
9093     B     True   False    True   False   False
9094     C     True   False   False    True   False
9095     D     True   False   False   False    True
9096     A     True    True   False   False   False
9097     B     True   False    True   False   False
9098     C     True   False   False    True   False
9099     D     True   False   False   False    True
9100     A     True    True   False   False   False
9101     B     True   False    True   False   False
9102     C     True   False   False    True   False
9103     D     True   False   False   False    True
9104     A     True    True   False   False   False
```

8.2 Determining deadlines

Source data:

- Project timetable defined in terms of business days allotted to complete each stage of the project.
- Start date of the project.

[6]:
```
project_start = '01 Jan 2018'
project_timetable = pd.DataFrame(data=[
                                       ['Development', 14],
                                       ['Acceptance', 2],
                                       ['Deployment', 3]
                                   ],
                                   columns=['Stage', 'Duration'])
project_timetable
```

[6]:
```
         Stage  Duration
0  Development        14
1   Acceptance         2
2   Deployment         3
```

The company works standard business hours. The country is Russia.

Task: Obtain the project deadlines as the calendar dates.

[7]:
```
clnd = clnd_ru
start_dates, end_dates = [], []
for stage_duration in project_timetable['Duration']:
    if not start_dates:
        start_dates = [clnd(project_start).rollforward()]
    else:
        start_dates.append(end_dates[-1] + 1)
    end_dates.append(start_dates[-1] + (stage_duration - 1))

project_timetable['Start'] = [day.to_timestamp() for day in start_dates]
project_timetable['Deadline'] = [day.to_timestamp() for day in end_dates]

project_timetable
```

[7]:
```
         Stage  Duration      Start   Deadline
0  Development        14 2018-01-09 2018-01-26
1   Acceptance         2 2018-01-29 2018-01-30
2   Deployment         3 2018-01-31 2018-02-02
```

Analysis

Two `timeboard` methods are used in this example:

- `Timeboard.get_workshift()` is called in line 5 disguised as `clnd(project_start)`. It puts the start date of the project into the context of the timeline of the calendar and returns the corresponding workshift.
- `Workshift.rollforward()` is called by name in line 5 and by proxy of operator + in lines 7 and 8.

When called without arguments (line 5) `rollforward()` returns the nearest on-duty workshift. In terms of our calendar, this means the nearest business day. It may be either *project_start* date itself or the next working day if *project_start* is a weekend or a holiday. In Russia, the first 8 days of January 2017 were holidays, hence, `clnd('01 Jan 2017').rollforward()` returns 09 Jan 2017.

When called with an integer argument, `rollforward(n)` moves *n* days toward the future skipping weekends and holidays. This is done in lines 7 and 8 where operator + is used as a shortcut for `rollforward`. For example, note that `rollforward(1)` called on Friday, 26 Jan 2017 (the end of Development stage), returns Monday, 29 Jan 2017, which becomes the start of Acceptance stage.

8.3 Generating shift schedule

Source data: timeboard of all shifts in a call center.

Task: generate the schedule of team's 'A' shifts for the week of 17 April 2017. (This is the same interval which illustrates the call center's timeboard in *Setting up the calendar* section above).

[8]:
```
clnd = clnd_cc
schedule = clnd.schedules['team_A']
period = clnd('17 April 2017', period='W')
```

(continues on next page)

(continued from previous page)

```
    shifts = [
        [ws.start_time, ws.duration, ws.end_time.ceil(clnd.base_unit_freq)]
        for ws in period.workshifts(schedule=schedule)
    ]

    pd.DataFrame(shifts, columns=['Start', 'Duration', 'End'])
```

[8]:

```
                Start  Duration                 End
0 2017-04-17 02:00:00         6 2017-04-17 08:00:00
1 2017-04-18 08:00:00        10 2017-04-18 18:00:00
2 2017-04-19 18:00:00         8 2017-04-20 02:00:00
3 2017-04-21 02:00:00         6 2017-04-21 08:00:00
4 2017-04-22 08:00:00        10 2017-04-22 18:00:00
5 2017-04-23 18:00:00         8 2017-04-24 02:00:00
```

Analysis

- `Timeboard.schedules` is a dictionary of schedules registered for our timeboard. In line 2 the schedule for team 'A' is retrieved.
- In line 3 `clnd('17 April 2017', period='A')` is a call of `Timeboard.get_interval()` returning an interval of shifts which belong to the calendar week 17-23 of April.
- `Interval.workshifts()` in line 6 returns a generator yielding all on-duty workshifts of the interval. Shifts are classified as "on duty" or "off duty" according to the schedule which is supplied to the method. By default, the method uses the default schedule of the timeboard. It would not be suitable for our purpose as under the default schedule every shift is on duty (the call center is always working). Hence we passed a specific schedule which selects only shifts labeled with 'A'.
- `start_time`, `duration`, and `end_time` are workshift attributes. We use `pandas.Timestamp.ceil()` to round up the end time of a workshift to the beginning of the next base unit of the timeboard.

8.4 Average annual headcount

The following examples are based on a fictitious company *Kings and Queens Ltd.*

Source data: staff register for *Kings and Queens Ltd.* containing for each employee:

- dates of entering and leaving the company
- salary rate

The value of `None` as the leaving date means that this person is still with the company.

[9]:

```
    staff = [
              ['Doran', '01 Feb 2012', '11 Nov 2017', 700],
              ['Robert', '10 May 2012', '01 Jan 2017', 1000],
              ['Joffrey', '03 Jan 2017', '17 Jul 2017', 800 ],
              ['Stannis', '02 Jan 2017', '07 Nov 2017', 500],
              ['Robb', '03 Apr 2017', '28 Apr 2017', 200],
              ['Daenerys', '18 Apr 2017', None, 500],
              ['Tommen', '18 Jul 2017', '29 Dec 2017', 800],
              ['Cersei', '30 Dec 2017', None, 1000],
              ['Jon', '01 Feb 2018', None, 100]
             ]
    register = pd.DataFrame(data=staff,
                            columns=['Name', 'Enter', 'Leave', 'Rate']).set_index(
↪'Name')
```

(continues on next page)

(continued from previous page)

```
    register
```

[9]:
```
                Enter        Leave  Rate
Name
Doran      01 Feb 2012  11 Nov 2017   700
Robert     10 May 2012  01 Jan 2017  1000
Joffrey    03 Jan 2017  17 Jul 2017   800
Stannis    02 Jan 2017  07 Nov 2017   500
Robb       03 Apr 2017  28 Apr 2017   200
Daenerys   18 Apr 2017         None   500
Tommen     18 Jul 2017  29 Dec 2017   800
Cersei     30 Dec 2017         None  1000
Jon        01 Feb 2018         None   100
```

Task: Calculate the average annual headcount of the company in 2017.

As an intermediate step, we will find out what portion of the year 2017 each person has worked for *Kings and Queens Ltd.* The data will be stored in the new column 'Worked_in_2017'.

[10]:
```
    clnd = clnd_uk

    y2017 = clnd('2017', period='A')
    register['Worked_in_2017'] = [
         clnd(tenure) / y2017 for tenure in zip(register.Enter, register.Leave)
    ]

    register
```

[10]:
```
                Enter        Leave  Rate  Worked_in_2017
Name
Doran      01 Feb 2012  11 Nov 2017   700        0.869048
Robert     10 May 2012  01 Jan 2017  1000        0.000000
Joffrey    03 Jan 2017  17 Jul 2017   800        0.539683
Stannis    02 Jan 2017  07 Nov 2017   500        0.857143
Robb       03 Apr 2017  28 Apr 2017   200        0.071429
Daenerys   18 Apr 2017         None   500        0.710317
Tommen     18 Jul 2017  29 Dec 2017   800        0.460317
Cersei     30 Dec 2017         None  1000        0.000000
Jon        01 Feb 2018         None   100        0.000000
```

Analysis

Two `timeboard` methods are used in this example:

- `Timeboard.get_interval()` is called in line 3 disguised as `clnd('2017', period='A')` and in line 5 as `clnd(tenure)`. The former call returns the interval which corresponds to the calendar year 2017. The latter call returns an interval which is bounded by the two dates supplied in *tenure* tuple: the day when the employee entered the company, and the day when he or she left.
- `Interval.what_portion_of()` is called by proxy of the division operator `/` in lines 5. This method finds out what portion of the second operand (the year 2017) is contained within the first operand (the working period of a person). Only business days are counted.

Note. For the employees still working at the company, the second element of *tenure* tuple is *None*. It means that the interval returned by `clnd(tenure)` extends until the last day of the calendar. The end time of the calendar is stored in `clnd.end_time` attribute.

The last step in calculating the average annual headcount in 2017 is trivial. We sum up all values in 'Worked_in_2017' column.

[11]:
```
    headcount = register.Worked_in_2017.sum()
    headcount
```

[11]:
```
3.5079365079365079
```

8.5 Calculating wages and salaries payable

Source data:

- Staff register with dates of entering and leaving the company and wage/salary rates.
- Pay period.

Task: For each employee determine the amount of wage/salary payable in the given pay period.

8.5.1 Periodic salary

Suppose salary is paid monthly. Below is the calculation of the salaries payable to the employees of *Kings and Queens Ltd.* in April 2017.

[12]:
```
    pay_period = clnd('April 2017', period='M')
    register['Salary_April'] = [
         clnd(tenure) / pay_period for tenure in zip(register.Enter, register.Leave)
    ] * register.Rate

    register
```

[12]:

Name	Enter	Leave	Rate	Worked_in_2017	Salary_April
Doran	01 Feb 2012	11 Nov 2017	700	0.869048	700.0
Robert	10 May 2012	01 Jan 2017	1000	0.000000	0.0
Joffrey	03 Jan 2017	17 Jul 2017	800	0.539683	800.0
Stannis	02 Jan 2017	07 Nov 2017	500	0.857143	500.0
Robb	03 Apr 2017	28 Apr 2017	200	0.071429	200.0
Daenerys	18 Apr 2017	None	500	0.710317	250.0
Tommen	18 Jul 2017	29 Dec 2017	800	0.460317	0.0
Cersei	30 Dec 2017	None	1000	0.000000	0.0
Jon	01 Feb 2018	None	100	0.000000	0.0

Analysis

The same methods are used as with calculating the values for `Worked_in_2017` column. Finally, the portion of the April 2017 taken by the tenure of each employee is multiplied by the salary rate of the employee.

Note that Daenerys has worked only a part of April 2017 - exactly a half (9 of 18 working days), therefore she is paid proportionally. However, Robb checked out all working days in the months because the first, the second, the 29th and the 30th of April - all fall on the weekends. Hence, Robb receives the full monthly salary.

8.5.2 Per-shift wage

Suppose that the staff of *Kings and Queens Ltd.* forms the team 'A' of the call center operators. The operators are paid 80 coins per shift. The task is to calculate the wages payable for a week of 17 April 2017.

(For your reference, the schedule of team's shifts for this week has been generated in an earlier example; it contains 6 shifts.)

[13]:
```
clnd = clnd_cc

pay_period = clnd('17 April 2017', period='W')
sdl_a = clnd.schedules['team_A']
shift_rate = 80

register['Wage_shifts'] = [
    clnd(tenure).overlap(pay_period).count(schedule=sdl_a) * shift_rate
    for tenure in zip(register.Enter, register.Leave)
]

register[['Enter', 'Leave', 'Wage_shifts']]
```

[13]:

Name	Enter	Leave	Wage_shifts
Doran	01 Feb 2012	11 Nov 2017	480
Robert	10 May 2012	01 Jan 2017	0
Joffrey	03 Jan 2017	17 Jul 2017	480
Stannis	02 Jan 2017	07 Nov 2017	480
Robb	03 Apr 2017	28 Apr 2017	480
Daenerys	18 Apr 2017	None	400
Tommen	18 Jul 2017	29 Dec 2017	0
Cersei	30 Dec 2017	None	0
Jon	01 Feb 2018	None	0

Analysis

Four *timeboard* methods are used in this example:

- `Timeboard.get_interval()` is called in lines 3 and 8. `clnd('17 April 2017', period='A')` returns an interval of shifts which belong to the calendar week 17-23 of April. `clnd(tenure)` returns the period of time when the person has had a job in the company.
- `Timeboard.schedules` is a dictionary of schedules registered for our timeboard. In line 4 the schedule for team 'A' is retrieved.
- `Interval.overlap()` is a part of the chain of methods in line 8. It returns the interval that is the intersection of two intervals: the employee's tenure and the pay period.
- `Interval.count()` is the last method called in line 8. It returns the number of on-duty workshifts in the given interval. Shifts are classified as "on duty" or "off duty" according to the schedule which is supplied to the method. By default, the method uses the default schedule of the timeboard. It would not be suitable for our purpose as under the default schedule every shift is on duty (the call center is always working). Hence we passed the specific schedule which selects only shifts labeled with 'A'.

8.5.3 Hourly pay

Let us change the pay scheme. Suppose the operators are paid 10 coins per hour. The task is the same: calculate the wages payable for a week of 17 April 2017.

[14]:
```
clnd = clnd_cc

pay_period = clnd('17 April 2017', period='W')
sdl_a = clnd.schedules['team_A']
hourly_rate = 10

register['Wage_hours'] = [
```

(continues on next page)

(continued from previous page)

```
        clnd(tenure).overlap(pay_period).worktime(schedule=sdl_a) * hourly_rate
        for tenure in zip(register.Enter, register.Leave)
    ]

    register[['Enter', 'Leave', 'Wage_shifts', 'Wage_hours']]
```

[14]:

Name	Enter	Leave	Wage_shifts	Wage_hours
Doran	01 Feb 2012	11 Nov 2017	480	480.0
Robert	10 May 2012	01 Jan 2017	0	0.0
Joffrey	03 Jan 2017	17 Jul 2017	480	480.0
Stannis	02 Jan 2017	07 Nov 2017	480	480.0
Robb	03 Apr 2017	28 Apr 2017	480	480.0
Daenerys	18 Apr 2017	None	400	420.0
Tommen	18 Jul 2017	29 Dec 2017	0	0.0
Cersei	30 Dec 2017	None	0	0.0
Jon	01 Feb 2018	None	0	0.0

Analysis

This snippet is analogous to the previous example. The only change is that in line 8, instead of `count()`, the last method called is `Interval.worktime()`. With this timeboard `worktime()` returns the total count of hours in all on-duty workshifts of the interval. As with `count()`, a schedule is passed to `worktime()` in order to tell on-duty shifts from off-duty ones.

8.6 Calculating bonus based on time worked

Source data:

- Staff register with dates of entering and leaving the company.
- Bonus coefficient.
- Bonus increment for each year worked.

The annual bonus is payable to the employees who have stayed in the company for a half of the year or more. The size of the bonus is the total annual salary multiplied by the bonus coefficient. The coefficient is increased for each full year spent with the company.

Task: For each employee calculate the bonus payable for the year 2017.

```
[15]:  # Housekeeping: remove the now irrelevant columns from the dataframe.
       register.drop(['Wage_shifts', 'Wage_hours'], axis=1, inplace=True)
```

Let's return *Kings and Queens Ltd.* to the standard office calendar.

As an intermediate step, we will find out how many years each employee has spent with the company by the end of the year 2017. The results will be stored in `Total_yrs` column.

```
[16]:  clnd = clnd_uk

       by_end_of_2017 = clnd((None, '31 Dec 2017'))
       register['Total_yrs'] = [
            clnd(tenure).overlap(by_end_of_2017).count_periods('A')
            for tenure in zip(register.Enter, register.Leave)
       ]
```

(continues on next page)

(continued from previous page)

```
    register
```

[16]:

```
                Enter        Leave  Rate  Worked_in_2017  Salary_April  \
Name
Doran      01 Feb 2012  11 Nov 2017   700        0.869048         700.0
Robert     10 May 2012  01 Jan 2017  1000        0.000000           0.0
Joffrey    03 Jan 2017  17 Jul 2017   800        0.539683         800.0
Stannis    02 Jan 2017  07 Nov 2017   500        0.857143         500.0
Robb       03 Apr 2017  28 Apr 2017   200        0.071429         200.0
Daenerys   18 Apr 2017         None   500        0.710317         250.0
Tommen     18 Jul 2017  29 Dec 2017   800        0.460317           0.0
Cersei     30 Dec 2017         None  1000        0.000000           0.0
Jon        01 Feb 2018         None   100        0.000000           0.0

           Total_yrs
Name
Doran       5.785714
Robert      4.646825
Joffrey     0.539683
Stannis     0.857143
Robb        0.071429
Daenerys    0.710317
Tommen      0.460317
Cersei      0.000000
Jon         0.000000
```

Analysis

This snippet reiterates the composition of the two previous examples. There are two modifications:

- `clnd((None, '31 Dec 2017'))` returns the interval from the beginning of the calendar to the day of 31 Dec 2017 inclusive. By passing this interval to `overlap()` called on the tenure in line 5 we effectively drop the part of the employee's tenure which extends into 2018 and beyond.
- The last method in the method chain in line 5 is `Interval.count_periods()` which calculates how many years fit into the interval. The method can see only business days. This is why the result for Cersei is zero in spite of the fact that she joined the company in 2017. Both 30 and 31 of December 2017 were days off, so she checked out no working days in 2017.

The rest of the code invokes the methods already made appearance in the earlier examples. In line 4 we select employees who are eligible for the bonus. In lines 6-10 their annual salaries for the year 2017 are calculated assuming that salary is paid monthly. In line 12 the bonus coefficient is incremented by taking into account the total number of years worked. In line 13 the resulting coefficient is applied to the annual salaries.

[17]:

```
    bonus_coefficient = 0.5
    bonus_increment_per_year = 0.1

    eligibles = register[register.Worked_in_2017 >= 0.5]

    y2017 = clnd('2017', period='A')
    annual_salary = [
         clnd(tenure).overlap(y2017).count_periods('M')
         for tenure in zip(eligibles.Enter, eligibles.Leave)
    ] * eligibles.Rate

    register['Bonus'] = (bonus_coefficient*(1 + bonus_increment_per_year*eligibles.
↪Total_yrs)) \
```

(continues on next page)

(continued from previous page)

```
                        * annual_salary

    register.Bonus = register.Bonus.fillna(0).round(2)
    register
```

[17]:

```
                Enter        Leave  Rate  Worked_in_2017  Salary_April  \
Name
Doran     01 Feb 2012  11 Nov 2017   700        0.869048         700.0
Robert    10 May 2012  01 Jan 2017  1000        0.000000           0.0
Joffrey   03 Jan 2017  17 Jul 2017   800        0.539683         800.0
Stannis   02 Jan 2017  07 Nov 2017   500        0.857143         500.0
Robb      03 Apr 2017  28 Apr 2017   200        0.071429         200.0
Daenerys  18 Apr 2017         None   500        0.710317         250.0
Tommen    18 Jul 2017  29 Dec 2017   800        0.460317           0.0
Cersei    30 Dec 2017         None  1000        0.000000           0.0
Jon       01 Feb 2018         None   100        0.000000           0.0

           Total_yrs    Bonus
Name
Doran       5.785714  5725.91
Robert      4.646825     0.00
Joffrey     0.539683  2750.36
Stannis     0.857143  2775.97
Robb        0.071429     0.00
Daenerys    0.710317  2275.94
Tommen      0.460317     0.00
Cersei      0.000000     0.00
Jon         0.000000     0.00
```

CHAPTER 9

Release Notes

9.1 timeboard 0.2.4

Release date: June 25, 2022

9.1.1 Resolved issues

- Fixed changed in import path for Iterables.
- Tested compatibility with Python 3.9, 3.10.

9.2 timeboard 0.2.3

Release date: May 01, 2020

9.2.1 Resolved issues

- Incompatibility with the breaking API changes introduced in pandas 1.0

9.2.2 Miscellaneous

- Russian business day calendar has been updated for 2020.

9.3 timeboard 0.2.2

Release date: May 01, 2019

9.3.1 Resolved issues

Breaking changes were introduced in pandas versions 0.23 and 0.24

- Pandas 0.23 moved *is_subperiod* function to another module
- Workaround for pandas issue #26258 (Adding offset to DatetimeIndex is broken)

9.4 timeboard 0.2.1

Release date: January 15, 2019

9.4.1 Miscellaneous

- Business day calendars for RU, UK, and US have been updated

9.5 timeboard 0.2

Release date: March 01, 2018

9.5.1 New features

- `Interval.overlap()` (also `*`) - return the interval that is the intersection of two intervals.
- `Interval.what_portion_of()` (also `/`) - calculate what portion of the other interval this interval takes up.
- `Interval.workshifts()` - return a generator that yields workshifts with the specified duty from the interval.
- Work time calculation: `Workshift.worktime()`, `Interval.worktime()`

9.5.2 Miscellaneous

- Performance: building any practical timeboard should take a fraction of a second.
- Documentation: added *Common Use Cases* section. It is also available as a `jupyter notebook`.

9.6 timeboard 0.1

Release date: February 01, 2018

This is the first release.

- genindex

Downloads:

- `jupyter notebook` with common use cases

Links:

- Github: https://github.com/mmamaev/timeboard
- PyPI: https://pypi.python.org/pypi/timeboard
- Documentation (this page): https://timeboard.readthedocs.io/