Businessdate Documentation

Release 0.6 [4 - Beta]

sonntagsgesicht, based on a fork of Deutsche Postbank [pbrisk

Saturday, 15 January 2022

CONTENTS

CHAPTER ONE

INTRODUCTION

1.1 Python library *businessdate*

A fast, efficient Python library for generating *business dates* for simple and fast date operations.

```
>>> from businessdate import BusinessDate

>>> BusinessDate(2017,12,31) + '2 weeks'
BusinessDate(20180114)

>>> BusinessDate(20171231) + '2w'  # same but shorter
BusinessDate(20180114)

>>> BusinessDate(20180114).to_date()
datetime.date(2018, 1, 14)
```

Typical banking business features are provided like *holiday adjustments* to move dates away from weekend days or *holidays*. As well as functionality to get *year fractions* depending on *day count conventions* as the lengths of interest payment periods.

Beside dates *business periods* can be created for time intervals like **10Y**, **3 Months** or **2b**. Those periods can easily be added to or subtracted from business dates.

Moreover *range* style *schedule generator* are provided to systematic build a list of dates. Such are used to set up a payment schedule of loan and financial derivatives.

1.2 Example Usage

```
>>> from datetime import date
>>> from businessdate import BusinessDate, BusinessPeriod


>>> BusinessDate(year=2014, month=1, day=11)
BusinessDate(20140111)

>>> BusinessDate(date(2014,1,11))
BusinessDate(20140111)

>>> BusinessDate(20140111)
BusinessDate(20140111)

>>> BusinessDate('20140111')
BusinessDate(20140111)

>>> BusinessDate('2015-12-31')
BusinessDate(20151231)

>>> BusinessDate('31.12.2015')
BusinessDate(20151231)

>>> BusinessDate('12/31/2015')
BusinessDate(20151231)

>>> BusinessDate(42369)
BusinessDate(20151231)

>>> BusinessDate(20140101) + BusinessPeriod('1Y3M')
BusinessDate(20150401)

>>> BusinessDate(20140101) + '1Y3M'
BusinessDate(20150401)

>>> BusinessDate(20170101) - '1Y1D'
BusinessDate(20151231)

>>> BusinessDate() == BusinessDate(date.today())
True

>>> BusinessDate('1Y3M20140101')
BusinessDate(20150401)
```

For more examples see the documentation.

1.3 Install

The latest stable version can always be installed or updated via pip:

```
$ pip install businessdate
```

1.4 Development Version

The latest development version can be installed directly from GitHub:

```
$ pip install --upgrade git+https://github.com/sonntagsgesicht/businessdate.git
```

or downloaded from https://github.com/sonntagsgesicht/businessdate.

1.5 ToDo

1. decide which base class or inheritance for *BusisnessDate* is better:
   a) *BaseDateFloat* (*float* inheritance)
   b) *BaseDateDatetimeDate* (*datetime.date* inheritance)

2. store businessdays adjustment convention and holidays as private property of *BusinessDate*. The information should not get lost under *BusinessPeriod* operation. Decide which date determines convention and holidays of a *BusinessRange*.

1.6 Contributions

Issues and Pull Requests are always welcome.

1.7 License

Code and documentation are available according to the Apache Software License (see LICENSE).

CHAPTER

TWO

TUTORIAL

To start with *businessdate* import it. Note that, since we work with dates, **datetime.date** might be useful, too. But not required. Nevertheless **datetime.date** is used inside *businessdate.businessdate.BusinessDate* from time to time.

```
>>> from datetime import date, timedelta
>>> from businessdate import BusinessDate, BusinessPeriod, BusinessRange,␣
↪BusinessSchedule
```

2.1 Creating Objects

2.1.1 BusinessDate

Once the library is loaded, creating business dates as simple as this.

```
>>> BusinessDate(year=2014, month=1, day=11)
BusinessDate(20140111)

>>> BusinessDate(date(2014,1,11))
BusinessDate(20140111)

>>> BusinessDate(20140111)
BusinessDate(20140111)

>>> BusinessDate('20140111')
BusinessDate(20140111)

>>> BusinessDate('2015-12-31')
BusinessDate(20151231)

>>> BusinessDate('31.12.2015')
BusinessDate(20151231)

>>> BusinessDate('12/31/2015')
BusinessDate(20151231)

>>> BusinessDate(42369) # number of days since January, 1st 1900
BusinessDate(20151231)
```

Even iterators like `list` or `tuple` work well.

```
>>> BusinessDate((20140216, 23011230, 19991111, 20200202))
(BusinessDate(20140216), BusinessDate(23011230), BusinessDate(19991111),␣
↪BusinessDate(20200202))
```

Much easier to generate container with periodical items is using *businessdate.businessrange.BusinessRange*.

By default an empty *businessdate.businessdate.BusinessDate* is initiated with the system date as given by +*datetime.date.today()**. To change this behavior: just set the *classattribute* businessdate.businessdate.BusinessDate.BASE_DATE to anything that can be understood as a business date, i.e. anything that meets *businessdate.businessdate.BusinessDate.is_businessdate()*.

```
>>> BusinessDate.BASE_DATE = '20110314'
>>> BusinessDate()
BusinessDate(20110314)

>>> BusinessDate.BASE_DATE = None
>>> BusinessDate().to_date() == date.today()
True
```

Attention: Setting *businessdate.businessdate.BusinessDate.BASE_DATE* to +*datetime.date.today()** is different to setting to **None** since +*datetime.date.today()** changes at midnight!

2.1.2 BusinessPeriod

There are two different categories of periods which can't be mixed.

One classical, given by a number of *years*, *month*, and *days*.

The second is *business days* or also known as working days, which are neither weekend days nor holidays. Holidays as seen as a container (e.g. *list* or *tuple*) of ` `datetime.date` which are understood as holidays.

Explicit keyword arguments of can be used to init an instance.

```
>>> BusinessPeriod()
BusinessPeriod('0D')

>>> BusinessPeriod(businessdays=10)
BusinessPeriod('10B')

>>> BusinessPeriod(years=2, months=6, days=1)
BusinessPeriod('2Y6M1D')

>>> BusinessPeriod(months=18)
BusinessPeriod('1Y6M')

>>> BusinessPeriod(years=1, months=6)
BusinessPeriod('1Y6M')
```

As seen *month* greater than 12 will be reduced to less or equal to 12 month with according years.

```
>>> BusinessPeriod(months=18)
BusinessPeriod('1Y6M')

>>> BusinessPeriod(years=2, months=6, days=1)
BusinessPeriod('2Y6M1D')
```

But this cannot be performed for days.

```
>>> BusinessPeriod(months=1, days=45)
BusinessPeriod('1M45D')
```

(continues on next page)

(continued from previous page)

```
>>> BusinessPeriod(months=2, days=14)
BusinessPeriod('2M14D')

>>> BusinessPeriod(months=2, days=15)
BusinessPeriod('2M15D')
```

Caution: As mentioned, classical period input arguments *years*, *month* and *days* must not be combined with *businessdays*.

```
>>> BusinessPeriod(businessdays=1, days=1)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "/Users/jph/Dropbox/Apps/GitHub/sonntagsgesicht/businessdate/businessdate/
↪businessperiod.py", line 103, in __init__
    raise ValueError("Either (years,months,days) or businessdays must be zero for
↪%s" % self.__class__.__name__)
ValueError: Either (years,months,days) or businessdays must be zero for␣
↪BusinessPeriod
```

Moreover, the difference of two instances of **datetime.date*+ or resp. a **datetime.timedelta** instance can be used to init, too.

```
>>> june_the_first, december_the_thirty_first = date(2010,6,1), date(2010,12,31)
>>> december_the_thirty_first-june_the_first
datetime.timedelta(days=213)

>>> BusinessPeriod(december_the_thirty_first-june_the_first)
BusinessPeriod('213D')

>>> timedelta(213)
datetime.timedelta(days=213)

>>> BusinessPeriod(timedelta(213))
BusinessPeriod('213D')
```

Similar to `businessdate.businessdate.BusinessDate` convenient string input work as well. Such a string represents again either periods of business days or classical periods.

```
>>> BusinessPeriod('0b')
BusinessPeriod('0D')

>>> BusinessPeriod('10D')
BusinessPeriod('10D')

>>> BusinessPeriod('1y3m4d')
BusinessPeriod('1Y3M4D')

>>> BusinessPeriod('18M')
BusinessPeriod('1Y6M')

>>> BusinessPeriod('1Q')
BusinessPeriod('3M')

>>> BusinessPeriod('2w')
```

(continues on next page)

(continued from previous page)

```
BusinessPeriod('14D')

>>> BusinessPeriod('10B')
BusinessPeriod('10B')
```

Inputs like **1Q** and **2W** work, too. Here **Q** stands for quarters, i.e. 3 months, and **W** for weeks, i.e. 7 days.

As a convention in financial markets these three additional shortcuts **ON** for *over night*, **TN** *tomorrow next* and **DD** *double days* exist.

```
>>> BusinessPeriod('ON')
BusinessPeriod('1B')

>>> BusinessPeriod('TN')
BusinessPeriod('2B')

>>> BusinessPeriod('DD')
BusinessPeriod('3B')
```

The `businessdate.businessperiod.BusinessPeriod` constructor understands even negative inputs. Please note the behavior of the preceding sign!

```
>>> BusinessPeriod('-0b')
BusinessPeriod('0D')

>>> BusinessPeriod('-10D')
BusinessPeriod('-10D')

>>> BusinessPeriod('-1y3m4d')
BusinessPeriod('-1Y3M4D')

>>> BusinessPeriod('-18M')
BusinessPeriod('-1Y6M')

>>> BusinessPeriod('-1Q')
BusinessPeriod('-3M')

>>> BusinessPeriod('-2w')
BusinessPeriod('-14D')

>>> BusinessPeriod('-10B')
BusinessPeriod('-10B')

>>> BusinessPeriod(years=-2, months=-6, days=-1)
BusinessPeriod('-2Y6M1D')
```

Caution: Beware of the fact that all non zero attributes must meet the same sign.

```
>>> BusinessPeriod(months=1, days=-1)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "/Users/jph/Dropbox/Apps/GitHub/sonntagsgesicht/businessdate/businessdate/
↪businessperiod.py", line 106, in __init__
    "(years, months, days)=%s must have equal sign for %s" % (str(ymd), self.__
↪class__.__name__))
ValueError: (years, months, days)=(0, 1, -1) must have equal sign for␣
↪BusinessPeriod
```

```
>>>
```

2.1.3 BusinessRange

Since *BusinessRange* just builds a periodical list of items like a *range* statement, it meets a similar signature and defaults.

```
>>> BusinessDate()
BusinessDate(20151225)

>>> start = BusinessDate(20151231)
>>> end = BusinessDate(20181231)
>>> rolling = BusinessDate(20151121)

>>> BusinessRange(start)
[BusinessDate(20151225), BusinessDate(20151226), BusinessDate(20151227),␣
↪BusinessDate(20151228), BusinessDate(20151229), BusinessDate(20151230)]

>>> BusinessRange(start) == BusinessRange(BusinessDate(), start, '1d', start)
True

>>> len(BusinessRange(start))
6

>>> len(BusinessRange(start)) == start.diff_in_days(end)
False

>>> BusinessDate() in BusinessRange(start)
True

>>> start not in BusinessRange(start)
True
```

To understand the rolling, think of periodical date pattern (like a wave) expanding from rolling date to future an past. Start and end date set boundaries such that all dates between them are in the business range.

If the start date meets those date, it is included. But the end date will never be included.

```
>>> start in BusinessRange(start, end, '1y', end)
True

>>> end in BusinessRange(start, end, '1y', end)
False

>>> BusinessRange(start, end, '1y', end)
[BusinessDate(20151231), BusinessDate(20161231), BusinessDate(20171231)]
```

If the start date does not meet any date in the range, it is not included.

```
>>> start in BusinessRange(start, end, '1y', rolling)
False

>>> end in BusinessRange(start, end, '1y', rolling)
False
```

(continues on next page)

(continued from previous page)

```
>>> BusinessRange(start, end, '1y', rolling)
[BusinessDate(20161121), BusinessDate(20171121), BusinessDate(20181121)]
```

Rolling on the same *start* and *end* but different *rolling* may lead to different ranges.

```
>>> start = BusinessDate(20150129)
>>> end = BusinessDate(20150602)
>>> rolling_on_start = BusinessRange(start, end, '1m1d', start)
>>> rolling_on_end = BusinessRange(start, end, '1m1d', end)

>>> rolling_on_start == rolling_on_end
False

>>> rolling_on_start
[BusinessDate(20150129), BusinessDate(20150301), BusinessDate(20150331),␣
↪BusinessDate(20150502)]

>>> rolling_on_end
[BusinessDate(20150129), BusinessDate(20150227), BusinessDate(20150331),␣
↪BusinessDate(20150501)]
```

Luckily, straight periods, e.g.

- annually,
- semi-annually,
- quarterly,
- monthly,
- weekly or
- daily,

don't mix-up in such a way.

```
>>> start = BusinessDate(20200202)
>>> end = start + BusinessPeriod('1y') * 10
>>> BusinessRange(start, end, '1y', start) == BusinessRange(start, end, '1y
↪', end)
True

>>> end = start + BusinessPeriod('6m') * 10
>>> BusinessRange(start, end, '6m', start) == BusinessRange(start, end, '6m
↪', end)
True

>>> end = start + BusinessPeriod('1q') * 10
>>> BusinessRange(start, end, '1q', start) == BusinessRange(start, end, '1q
↪', end)
True

>>> end = start + BusinessPeriod('1m') * 10
>>> BusinessRange(start, end, '1m', start) == BusinessRange(start, end, '1m
↪', end)
True

>>> end = start + BusinessPeriod('1w') * 10
>>> BusinessRange(start, end, '1w', start) == BusinessRange(start, end, '1w
↪', end)
```

(continues on next page)

(continued from previous page)

```
True

>>> end = start + BusinessPeriod('1d') * 10
>>> BusinessRange(start, end, '1d', start) == BusinessRange(start, end, '1d
↪', end)
True
```

2.1.4 BusinessSchedule

A *businessdate.businessschedule.BusinessSchedule*, as inhereted from *businessdate.businessrange.BusinessRange*, provides nearly the same features as *businessdate.businessrange.BusinessRange*. But *businessdate.businessschedule.BusinessSchedule* lists contain always start date and end date!

Since the first as well as the last period can be very short (short stubs), they can be trimmed to give a first and/or last period as long stubs.

```
>>> start = BusinessDate(20151231)
>>> end = BusinessDate(20181231)
>>> rolling = BusinessDate(20151121)

>>> BusinessRange(start, end, '1y', rolling)
[BusinessDate(20161121), BusinessDate(20171121), BusinessDate(20181121)]

>>> BusinessSchedule(start, end, '1y', rolling)
[BusinessDate(20151231), BusinessDate(20161121), BusinessDate(20171121),␣
↪BusinessDate(20181121), BusinessDate(20181231)]

>>> BusinessSchedule(start, end, '1y', rolling).first_stub_long()
[BusinessDate(20151231), BusinessDate(20171121), BusinessDate(20181121),␣
↪BusinessDate(20181231)]

>>> BusinessSchedule(start, end, '1y', rolling).last_stub_long()
[BusinessDate(20151231), BusinessDate(20161121), BusinessDate(20171121),␣
↪BusinessDate(20181231)]

>>> BusinessSchedule(start, end, '1y', rolling).first_stub_long().last_stub_long()
[BusinessDate(20151231), BusinessDate(20171121), BusinessDate(20181231)]
```

2.1.5 BusinessHolidays

Since we deal with *businessdate.businessdate.BusinessDate* the container class *businessdate.businessholidays.BusinessHolidays* is useful as it converts nearly anything input into **datetime.date**.

Provide list of **datetime.date** or anything having attributes *year*, *month* and *days*, e.g. iterable that yields of *businessdate.businessdate.BusinessDate*.

For example you can use projects like python-holidays or workcalendar which offer holidays in many different countries, regions and calendars.

Build-in are *businessdate.businessholidays.TargetHolidays* which are bank holidays in euro banking system TARGET.

They serve as default value if no holidays are given. They can be changed on demand via the class attribute **DEFAULT_HOLIDAYS** in *businessdate.businessdate.BusinessDate*.

```
>>> BusinessDate(20100101) in BusinessDate.DEFAULT_HOLIDAYS
True

>>> BusinessDate.DEFAULT_HOLIDAYS = list()
>>> BusinessDate(20100101) in BusinessDate.DEFAULT_HOLIDAYS
False
```

2.2 Calculating Dates and Periods

Attention: Even *adding* and *subtracting* Dates and Periods suggest to be a kind of algebraic operation like adding and subtracting numbers. But they are not, at least not in a similar way!

Algebraic operations of numbers are known to be

- compatible, e.g. $3 + 3 = 2 * 3 = 2 + 2 + 2$
- associative, e.g. $(1 + 2) + 3 = 1 + (2 + 3)$
- distributive, e.g. $(1 + 1) * 2 = 2 + 2$
- commutative, e.g. $1 + 2 = 2 + 1$

Due to different many days in different months as well as leap years periods do not act that way on dates.

Note: For example, add 2 month to March, 31th should give May, 31th. But adding 2 times 1 month will give May, 30th, since

March, 31th + 1 month = April, 30th

April, 30th + 1 month = May, 30th

Even more pitfalls exist when izt comes to calculate dates and calendars. Fortunately periods acting on them self behave much more like numbers.

All this is build into *businessdate.businessperiod.BusinessPeriod* and *businessdate.businessdate.BusinessDate*.

2.2.1 Adding

2.2.1.1 Date + Period

Adding two dates does not any sense. So we can only add a period to a date to give a new date

```
>>> BusinessDate(20150612) + BusinessPeriod('6M19D')
BusinessDate(20151231)
```

2.2.1.2 Period + Period

And two periods to give a new period - as long as the do not mix business days and classical periods.

```
>>> BusinessPeriod('6M10D') + BusinessPeriod('9D')
BusinessPeriod('6M19D')

>>> BusinessPeriod('9D') + BusinessPeriod('6M10D')
BusinessPeriod('6M19D')

>>> BusinessPeriod('5B') + BusinessPeriod('10B')
BusinessPeriod('15B')
```

2.2.2 Subtracting

2.2.2.1 Date - Date

Surprisingly, the difference of two dates makes sense, as the distance in numeber of years than months and finaly days from the early to the later date.

```
>>> BusinessDate(20151231) - BusinessDate(20150612)
BusinessPeriod('6M19D')
```

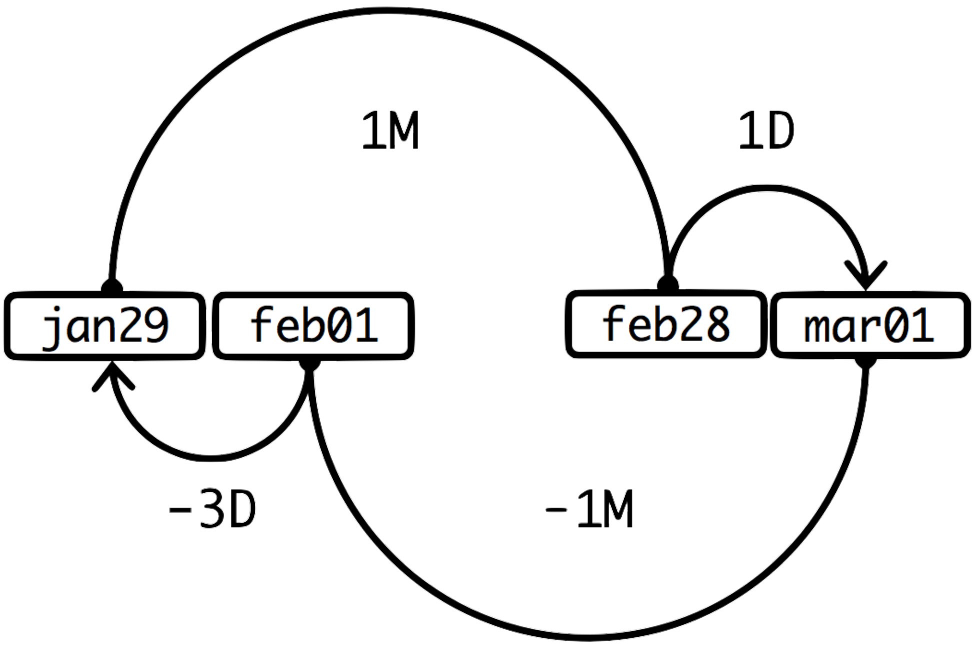
Those are just the inverse operations

```
>>> period = BusinessDate(20151231) - BusinessDate(20150612)
>>> BusinessDate(20151231) == BusinessDate(20150612) + period
True
```

But note that these operations are not commutative, i.e. swapping the order can give something completely different as the the direction of the point of view is changed.

```
>>> dec31 = BusinessDate(20151231)
>>> jun12 = BusinessDate(20150612)

>>> dec31 - jun12  # it takes 6 months and 19 days from jun12 to dec31
BusinessPeriod('6M19D')

>>> jun12 - dec31  # jun12 is 6 months and 18 days before dec31
BusinessPeriod('-6M18D')

>>> jan29 = BusinessDate(20150129)
>>> mar01 = BusinessDate(20150301)

>>> mar01 - jan29  # from jan29 yoe waits 1 month and 1 day until mar01
BusinessPeriod('1M1D')

>>> jan29 - mar01  # but mar01 was 1 month and 3 days before
BusinessPeriod('-1M3D')
```

This becomes clear if you check this with your calendar.



But still we get

```
>>> BusinessDate(20150612) - BusinessDate(20151231)
BusinessPeriod('-6M18D')

>>> BusinessDate(20150612) == BusinessDate(20151231) - BusinessPeriod('6M18D
↪')
True
```

2.2.2.2 Date - Period

And again, we can subtract a period from a date to give a new date.

```
>>> BusinessDate(20151231) - BusinessPeriod('6M18D') ==␣
↪BusinessDate(20150612)
True

>>> BusinessDate(20151231) - BusinessPeriod('10b')
BusinessDate(20151216)
```

2.2.2.3 Period + Period

And straight forward, two periods substracted from each other to give a new period. Again, as long as the do not mix business days and classical periods.

```
>>> BusinessPeriod('6M19D') - BusinessPeriod('6M10D')
BusinessPeriod('9D')

>>> BusinessPeriod('-6M10D') - BusinessPeriod('-6M19D')
BusinessPeriod('9D')

>>> BusinessPeriod('10b') - BusinessPeriod('15b')
BusinessPeriod('-5B')
```

2.2.3 Multiplying

2.2.3.1 Period * int

Since an instance of a `BusinessPeriod` stored the number of *years*, *month*, *days* or *businessdays* as `int` one multiply this by integer, too.

Note that the number of *month* can be reduced if it's exceeds the number of 12. But we can not do anything like this with days.

```
>>> BusinessPeriod('1y2m3d') * 2
BusinessPeriod('2Y4M6D')

>>> BusinessPeriod('1y8m200d') * 2
BusinessPeriod('3Y4M400D')

>>> y, m, d = 1, 2, 3
>>> BusinessPeriod(years=y, months=m, days=d) * 2 ==␣
↪BusinessPeriod(years=y*2, months=m*2, days=d*2)
True

>>> BusinessPeriod('1y2m3d') * 2 == 2 * BusinessPeriod('1y2m3d')
True
```

2.2.4 Comparing

2.2.4.1 Dates

Calendars assume time to be evolving in strictly one direction, from past to future. Hence days can be well ordered and so be compared. Same for `BusinessDate`.

```
>>> BusinessDate(20151231) < BusinessDate(20160101)
True

>>> BusinessDate(20151231) == BusinessDate(20160101)
False

>>> BusinessDate(20151231) > BusinessDate(20160101)
False
```

2.2.4.2 Periods

Two Tuples of three numbers *(a,b,c)* and *(d,e,f)* have only a natural order if all three numbers meet the same relation, e.g.

(a,b,c) < *(d,e,f)* if $a < d$ and $b < e$ and $c < f$

(a,b,c) == *(d,e,f)* if $a == d$ and $b == e$ and $c == f$

In case of a two classical period as a *(years, months, days)* the problem can be reduced by comparing only two numbers *(years*12 + months, days)*.

But leveraging the order of dates, a period p can be seen as greater than a period q if for any possible date d adding both periods give always the same resulting order in dates.

I.e. we get

$p < q$ if $d + p < d + q$ for all dates d

Hence, we are left with only *few* situations, which might give for different dates d and *d'* *d + p < d + q* but *d' + p >= d' + q*

Since *days* vary in different month, periods close to each other are difficult to compare, e.g. is **1M1D** greater or equal **31D**?

```
>>> p = BusinessPeriod('1M1D')
>>> q = BusinessPeriod('31D')

>>> BusinessDate(20150131) + p < BusinessDate(20150131) + q
True

>>> BusinessDate(20150731) + p < BusinessDate(20150731) + q
False
```

So, let *(a,b,c)* and *(d,e,f)* be two periods with

*m = (a - b) * 12 + b - e* and *d = c - f*

as the distance of both measured in *months* and *days*.

The sequence of the number of days in a period of given months with minimal days as well as max can be derived. The first 13 months listed.

months	num days
1	28 … 31
2	59 … 62
3	89 … 92
4	120 … 123
5	150 … 153
6	181 … 184
7	212 … 215
8	242 … 245
9	273 … 276
10	303 … 306
11	334 … 337
12	365 … 366
13	393 … 397

For those pairs of month and days any comparison of **<** or **>** is not well defined. Hence,

```
>>> BusinessPeriod('13m') < BusinessPeriod('392d')
False

>>> BusinessPeriod('13m') < BusinessPeriod('393d') # not well defined ->␣
↪None

>>> BusinessPeriod('13m') < BusinessPeriod('397d') # not well defined ->␣
↪None

>>> BusinessPeriod('13m') < BusinessPeriod('398d')
True
```

But

```
>>> BusinessPeriod('13m') <= BusinessPeriod('392d')
False

>>> BusinessPeriod('13m') <= BusinessPeriod('393d') # not well defined ->␣
↪None
```

(continues on next page)

(continued from previous page)

```
>>> BusinessPeriod('13m') <= BusinessPeriod('397d')
True

>>> BusinessPeriod('13m') <= BusinessPeriod('398d')
True
```

So comparison of arbitrary instances or BusinessPeriod only works for ==.

```
>>> BusinessPeriod('ON') == BusinessPeriod('1B')
True

>>> BusinessPeriod('7D') == BusinessPeriod('1W')
True

>>> BusinessPeriod('30D') == BusinessPeriod('1M')
False

>>> BusinessPeriod('1D') == BusinessPeriod('1B')
False
```

2.2.5 Adjusting

2.2.5.1 Dates

When adding a period to a date results on a weekend day may make no sense in terms of business date. This happens frequently when a interst payment plan is rolled out. In such a case all dates which fall either on weekend days or on holidays have to be moved (*adjusted*) to a business day.

In financial markets different conventions of business day adjustments are kown. Most of them are part of the ISDA Definitions which are not open to public. But see date rolling for more details.

```
>>> weekend_day = BusinessDate(20141129)
>>> weekend_day.weekday()  # Monday is 0 and Sunday is 6
5

>>> weekend_day.adjust('follow')  # move to next business day
BusinessDate(20141201)

>>> weekend_day.adjust('previous')  # move to previous business day
BusinessDate(20141128)

>>> weekend_day.adjust('mod_follow')  # move to next business day in same month else␣
↪pervious
BusinessDate(20141128)

>>> BusinessDate(20141122).adjust('mod_follow')  # move to next business day in same␣
↪month else pervious
BusinessDate(20141124)

>>> weekend_day.adjust('mod_previous')  # move to previous business day in same month␣
↪else follow
BusinessDate(20141128)

>>> weekend_day.adjust('start_of_month')  # move to first business day in month
```

(continues on next page)

(continued from previous page)

```
BusinessDate(20141103)

>>> weekend_day.adjust('end_of_month')  # move to last business day in month
BusinessDate(20141128)
```

In order to provide specific holidays a list of **datetime.date** objects can be given as an extra argument. It can convenient to use a *businessdate.businessholidays.BusinessHolidays* instance instead but any type that implements *__contain__* will work.

```
>>> weekend_day.adjust('follow', holidays=[BusinessDate(20141201)])  # move to next␣
↪business day
BusinessDate(20141202)
```

If no holidays are given the **DEFAULT_HOLIDAYS** of *businessdate.businessdate.BusinessDate* are used. By default those are the *TARGET holidays*.

To view all possible *convention* key words see *businessdate.businessdate.BusinessDate.adjust()* documentation.

Beside *businessdate.businessdate.BusinessDate* there is also *businessdate.businessrange.BusinessRange.adjust()* (same for *businessdate.businessschedule.BusinessSchedule*) which adjust all items in the *businessdate.businessrange.BusinessRange*.

```
>>> start = BusinessDate(20151231)
>>> BusinessRange(start)
[BusinessDate(20151225), BusinessDate(20151226), BusinessDate(20151227),␣
↪BusinessDate(20151228), BusinessDate(20151229), BusinessDate(20151230)]

>>> BusinessRange(start).adjust('mod_follow')
[BusinessDate(20151228), BusinessDate(20151228), BusinessDate(20151228),␣
↪BusinessDate(20151228), BusinessDate(20151229), BusinessDate(20151230)]
```

2.2.6 Measuring

2.2.6.1 Periods

Interest rates are agree and settled as annual rate. In contrast to this annual definition, interest payments are often semi-annually, quarterly or monthly or even daily.

In order to calculate an less than annal interest payment from an annual interest rate the *year fraction* of each particular period is used as

$$\textit{interest payment} = \textit{annual interest rate} * \textit{year fraction} * \textit{notional}$$

The *year fraction* depends on the days between the *start date* and *end date* of a period. In order to simplify calculation in the past there various financial markets convention to count days between dates, see detail on day count conventions.

The most common *day count conventions*, i.e. *year fraction*, are available by *businessdate.businessdate.BusinessDate.get_day_count()* and *businessdate.businessdate.BusinessDate.get_year_fraction()* (different name but same fuctionality).

To view all possible *convention* see *businessdate.businessdate.BusinessDate.get_day_count()* documentation.

```
>>> start_date = BusinessDate(20190829)
>>> end_date = start_date + '3M'

>>> start_date.get_day_count(end_date, 'act_act')
```

(continues on next page)

(continued from previous page)

```
0.25205479452054796

>>> start_date.get_day_count(end_date, 'act_36525')
0.2518822724161533

>>> start_date.get_day_count(end_date, 'act_365')
0.25205479452054796

>>> start_date.get_day_count(end_date, 'act_360')
0.25555555555555554

>>> start_date.get_day_count(end_date, '30_360')
0.25

>>> start_date.get_day_count(end_date, '30E_360')
0.25

>>> start_date.get_day_count(end_date, '30E_360_I')
0.25
```

2.3 BusinessDate Details

2.3.1 More Creation Patterns

More complex creation pattern work, too. They combine the creation of a date plus a period with business day adjustemnt conventions at start and/or end of the period.

Create an instance directly from a period or period string.

```
>>> BusinessDate()
BusinessDate(20161009)

>>> BusinessDate() + '1m'
BusinessDate(20161109)

>>> BusinessDate(BusinessPeriod(months=1))
BusinessDate(20161109)

>>> BusinessDate('1m')
BusinessDate(20161109)

>>> BusinessDate('15b')
BusinessDate(20161028)

>>> BusinessDate() + '15b'
BusinessDate(20161028)
```

This works with additional date, too.

```
>>> BusinessDate('1m20161213')
BusinessDate(20170113)

>>> BusinessDate('20161213') + '1m'
BusinessDate(20170113)
```

Adding the adjustment convention 'end_of_month' with a business date gives the following.

```
>>> BusinessDate('0bEOM')
BusinessDate(20161031)

>>> BusinessDate('EOM')
BusinessDate(20161031)

>>> BusinessDate().adjust('EOM')
BusinessDate(20161031)

>>> BusinessDate('15bEOM')
BusinessDate(20161121)

>>> BusinessDate().adjust('EOM') + '15b'
BusinessDate(20161121)
```

Adding the adjustment convention 'mod_follow' with a business date lead to this.

```
>>> BusinessDate('0bModFlw')
BusinessDate(20161010)

>>> BusinessDate('ModFlw')
BusinessDate(20161010)

>>> BusinessDate().adjust('ModFlw')
BusinessDate(20161010)

>>> BusinessDate('15bModFlw')
BusinessDate(20161031)

>>> BusinessDate().adjust('ModFlw') + '15b'
BusinessDate(20161031)
```

But a adjustment convention with a classical period and without a business date is ignored since the adjustment statement is ambiguous:

Should the start date (spot) or end date be adjusted?

```
>>> BusinessDate('1mEOM')
BusinessDate(20161109)

>>> BusinessDate('1mModFlw')
BusinessDate(20161109)
```

Adding zero business days clarifies it!

```
>>> BusinessDate('0b1mModFlw')
BusinessDate(20161110)

>>> BusinessDate('0b1mModFlw') == BusinessDate().adjust('ModFlw') + '1m'
True

>>> BusinessDate('1m0bModFlw')
BusinessDate(20161109)

>>> BusinessDate('1m0bModFlw') == (BusinessDate() + '1m').adjust('ModFlw')
True
```

Clearly business days may be non zero, too.

```
>>> BusinessDate('15b1mModFlw')
BusinessDate(20161130)

>>> BusinessDate('15b1mModFlw') == BusinessDate('ModFlw') + '15b' + '1m'
True

>>> BusinessDate('1m5bModFlw')
BusinessDate(20161116)

>>> BusinessDate('1m5bModFlw') == BusinessDate('1m').adjust('ModFlw') + '5b'
True
```

Putting all together we get.

```
>>> BusinessDate('15b1m5bModFlw20161213')
BusinessDate(20170213)

>>> bd = BusinessDate(20161213)
>>> bd = bd.adjust('ModFlw')
>>> bd = bd + '15b'
>>> bd = bd + '1m'
>>> bd = bd.adjust('ModFlw')
>>> bd = bd + '5b'
>>> bd
BusinessDate(20170213)

>>> BusinessDate('15b1m5bModFlw20161213') == (BusinessDate(20161213).adjust('ModFlw')␣
↪+ '15b' + '1m').adjust('ModFlw') + '5b'
True
```

2.3.2 BusinessDate Inheritance

Finally some lines on *base classes* *businessdate.basedate.BaseDateFloat* backed by **float** ...

```
>>> from datetime import date
>>> from businessdate.basedate import BaseDateFloat

>>> BaseDateFloat(40123.)
40123.0

>>> BaseDateFloat.from_ymd(2009, 11, 6)
40123.0

>>> BaseDateFloat.from_date(date(2009, 11, 6))
40123.0

>>> BaseDateFloat.from_float(40123.)
40123.0

>>> d = BaseDateFloat(40123.)
>>> d.year, d.month, d.day
(2009, 11, 6)

>>> d.to_ymd()
(2009, 11, 6)
```

(continues on next page)

(continued from previous page)

```
>>> d.to_date()
datetime.date(2009, 11, 6)

>>> d.to_float()
40123.0
```

… and *businessdate.basedate.BaseDateDatetimeDate* backed by **datetime.date**.

```
>>> from datetime import date
>>> from businessdate.basedate import BaseDateDatetimeDate

>>> BaseDateDatetimeDate(2009, 11, 6)
BaseDateDatetimeDate(2009, 11, 6)

>>> BaseDateDatetimeDate.from_ymd(2009, 11, 6)
BaseDateDatetimeDate(2009, 11, 6)

>>> BaseDateDatetimeDate.from_date(date(2009, 11, 6))
BaseDateDatetimeDate(2009, 11, 6)

>>> BaseDateDatetimeDate.from_float(40123.)
BaseDateDatetimeDate(2009, 11, 6)

>>> BaseDateDatetimeDate(2009, 11, 6)
BaseDateDatetimeDate(2009, 11, 6)

>>> d.year, d.month, d.day
(2009, 11, 6)

>>> d.to_ymd()
(2009, 11, 6)

>>> d.to_date()
datetime.date(2009, 11, 6)

>>> d.to_float()
40123.0
```

CHAPTER THREE

PROJECT DOCUMENTATION

BusinessDate	date class to perform calculations coming from financial businesses
`BusinessPeriod`	class to store and calculate date periods as combinations of days, weeks, years etc.
`BusinessRange`	class to build list of business days
`BusinessSchedule`	class to build date schedules incl start and end date
`BusinessHolidays`	holiday calendar class

3.1 Business Object Classes

3.1.1 BusinessDate

class `businessdate.businessdate.`**`BusinessDate`**(*year=None*, *month=0*, *day=0*, *convention=None*, *holidays=None*, *day_count=None*)

Bases: *businessdate.basedate.BaseDateDatetimeDate*

date class to perform calculations coming from financial businesses

Parameters

- **year** – number of year or some other input value t o create *BusinessDate* instance. When applying other input, this can be either `int`, `float`, `datetime.date` or `string` which will be parsed and transformed into equivalent `tuple` of `int` items *(year,month,day)* (See *tutorial* for details).
- **month** (*int*) – number of month in year 1 ... 12 (default: 0, required to be 0 when other input of year is used)
- **days** (*int*) – number of day in month 1 ... 31 (default: 0, required to be 0 when other input of year is used)

For all input arguments exits read only properties.

`ADJUST = 'No'`

`BASE_DATE = None`

`DATE_FORMAT = '%Y%m%d'`

`DAY_COUNT = 'act_36525'`

`DEFAULT_CONVENTION`(*holidays=()*)
does no adjustment.

`DEFAULT_HOLIDAYS = []`

`DEFAULT_DAY_COUNT`(*end*)
implements Act/365.25 Day Count Convention

classmethod is_businessdate(*d*)
checks whether the provided input can be a date

is_leap_year()
returns *True* for leap year and False otherwise

days_in_year()
returns number of days in the given calendar year

days_in_month()
returns number of days for the month

end_of_month()
returns the day of the end of the month as `BusinessDate` object

end_of_quarter()
returns the day of the end of the quarter as `BusinessDate` object

is_business_day(*holidays=None*)
returns *True* if date falls neither on weekend nor is in holidays (if given as container object)

add_period(*period_obj*, *holidays=None*)
adds a `BusinessPeriod` object or anythings that create one and returns `BusinessDate` object.

It is simply adding the number of *years*, *months* and *days* or if *businessdays* given the number of business days, i.e. days neither weekend nor in holidays (see also `BusinessDate.is_business_day()`)

diff_in_days(*end_date*)
calculates the distance to a `BusinessDate` in days

diff_in_ymd(*end_date*)

get_day_count(*end=None*, *day_count=None*)
counts the days as a year fraction to given date following the specified convention.

For more details on the conventions see module `businessdate.daycount`.

In order to get the year fraction according a day count convention provide one of the following convention key words:

- `30_360` implements 30/360 Day Count Convention.
- `30360` implements 30/360 Day Count Convention.
- `thirty360` implements 30/360 Day Count Convention.
- `30e_360` implements the 30E/360 Day Count Convention.
- `30e360` implements the 30E/360 Day Count Convention.
- `thirtye360` implements the 30E/360 Day Count Convention.
- `30e_360_i` implements the 30E/360 I. Day Count Convention.
- `30e360i` implements the 30E/360 I. Day Count Convention.
- `thirtye360i` implements the 30E/360 I. Day Count Convention.
- `act_360` implements Act/360 day count convention.
- `act360` implements Act/360 day count convention.
- `act_365` implements Act/365 day count convention.
- `act365` implements Act/365 day count convention.
- `act_36525` implements Act/365.25 Day Count Convention
- `act_365.25` implements Act/365.25 Day Count Convention
- `act36525` implements Act/365.25 Day Count Convention
- `act_act` implements Act/Act day count convention.

- `actact` implements Act/Act day count convention.

`get_year_fraction`(*end=None*, *day_count=None*)
wrapper for `BusinessDate.get_day_count()` method for different naming preferences

`adjust`(*convention=None*, *holidays=None*)
returns an adjusted `BusinessDate` if it was not a business day following the specified convention.

For details on business days see `BusinessDate.is_business_day()`.

For more details on the conventions see module `businessdate.conventions`

In order to adjust according a business day convention provide one of the following convention key words:

- `no` does no adjustment.
- `previous` adjusts to Business Day Convention "Preceding".
- `prev` adjusts to Business Day Convention "Preceding".
- `prv` adjusts to Business Day Convention "Preceding".
- `mod_previous` adjusts to Business Day Convention "Modified Preceding".
- `modprevious` adjusts to Business Day Convention "Modified Preceding".
- `modprev` adjusts to Business Day Convention "Modified Preceding".
- `modprv` adjusts to Business Day Convention "Modified Preceding".
- `follow` adjusts to Business Day Convention "Following".
- `flw` adjusts to Business Day Convention "Following".
- `modified` adjusts to Business Day Convention "Modified [Following]".
- `mod_follow` adjusts to Business Day Convention "Modified [Following]".
- `modfollow` adjusts to Business Day Convention "Modified [Following]".
- `modflw` adjusts to Business Day Convention "Modified [Following]".
- `start_of_month` adjusts to Business Day Convention "Start of month", i.e. first business day.
- `startofmonth` adjusts to Business Day Convention "Start of month", i.e. first business day.
- `som` adjusts to Business Day Convention "Start of month", i.e. first business day.
- `end_of_month` adjusts to Business Day Convention "End of month", i.e. last business day.
- `endofmonth` adjusts to Business Day Convention "End of month", i.e. last business day.
- `eom` adjusts to Business Day Convention "End of month", i.e. last business day.
- `imm` adjusts to Business Day Convention of "International Monetary Market".
- `cds_imm` adjusts to Business Day Convention "Single Name CDS".
- `cdsimm` adjusts to Business Day Convention "Single Name CDS".
- `cds` adjusts to Business Day Convention "Single Name CDS".

3.1.1.1 BusinessDate Base Classes

businessdate.businessdate.BusinessDate inherits from one of two possible base classes. One itself inherited by a native **float** class. The other inherited from **datetime.date** class.

Both classes are implemented to offer future releases the flexibility to switch from one super class to another if such offers better performance.

Currently *businessdate.businessdate.BusinessDate* inherits from *businessdate.basedate.BaseDateDatetimeDate* which offers more elaborated functionality.

Warning: Future releases of *businessdate* may be backed by different base classes.

`class businessdate.basedate.`**`BaseDateFloat`**`(`*x=0*`)`

Bases: `float`

native `float` backed base class for a performing date calculations counting days since Jan, 1st 1900

`property day`

`property month`

`property year`

`weekday``()`

`classmethod from_ymd``(`*year, month, day*`)`

creates instance from a `tuple` of `int` items *(year, month, day)*

`classmethod from_date``(`*d*`)`

creates instance from a `datetime.date` object *d*

`classmethod from_float``(`*x*`)`

creates from a `float` *x* counting the days since Jan, 1st 1900

`to_ymd``()`

returns the `tuple` of `int` items *(year, month, day)*

`to_date``()`

returns *datetime.date(year, month, day)*

`to_float``()`

returns `float` counting the days since Jan, 1st 1900

`class businessdate.basedate.`**`BaseDateDatetimeDate`**

Bases: `datetime.date`

`datetime.date` backed base class for a performing date calculations

`classmethod from_ymd``(`*year, month, day*`)`

creates instance from a `tuple` of `int` items *(year, month, day)*

`classmethod from_date``(`*d*`)`

creates instance from a `datetime.date` object *d*

`classmethod from_float``(`*x*`)`

creates from a `float` *x* counting the days since Jan, 1st 1900

`to_ymd``()`

returns the `tuple` of `int` items *(year, month, day)*

`to_date``()`

returns *datetime.date(year, month, day)*

`to_float``()`

returns `float` counting the days since Jan, 1st 1900

`to_serializable``(`**args, **kwargs*`)`

3.1.2 BusinessPeriod

`class businessdate.businessperiod.`**`BusinessPeriod`**(*period=''*, *years=0*, *quarters=0*, *months=0*, *weeks=0*, *days=0*, *businessdays=0*)

Bases: `object`

class to store and calculate date periods as combinations of days, weeks, years etc.

Parameters

- **`period`** (*`str`*) – encoding a business period. Such is given by a sequence of digits as `int` followed by a `char` - indicating the number of years **Y**, quarters **Q** (which is equivalent to 3 month), month **M**, weeks **W** (which is equivalent to 7 days), days **D**, business days **B**. E.g. **1Y2W3D** what gives a period of 1 year plus 2 weeks and 3 days (see *tutorial* for details).
- **`years`** (*`int`*) – number of years in the period (equivalent to 12 months)
- **`quarters`** (*`int`*) – number of quarters in the period (equivalent to 3 months)
- **`months`** (*`int`*) – number of month in the period
- **`weeks`** (*`int`*) – number of weeks in the period (equivalent to 7 days)
- **`days`** (*`int`*) – number of days in the period
- **`businessdays`** (*`int`*) – number of business days, i.e. days which are neither weekend nor `holidays`, in the period. Only either *businessdays* or the others can be given. Both at the same time is not allowed.

`property` **`years`**

`property` **`months`**

`property` **`days`**

`property` **`businessdays`**

`classmethod` **`is_businessperiod`**(*period*)

returns true if the argument can be understood as *`BusinessPeriod`*

`max_days`()

`min_days`()

3.1.3 BusinessSchedule

`class businessdate.businessschedule.`**`BusinessSchedule`**(*start*, *end*, *step*, *roll=None*)

Bases: *`businessdate.businessrange.BusinessRange`*

class to build date schedules incl start and end date

Parameters

- **`start`** (`BusinessDate`) – start date of schedule
- **`end`** (`BusinessDate`) – end date of schedule
- **`step`** (`BusinessPeriod`) – period distance of two dates
- **`roll`** (`BusinessDate`) – origin of schedule

convenient class to build date schedules a schedule includes always start and end date and rolls on roll, i.e. builds a sequence by adding and/or substracting step to/from roll. start and end slice the relevant dates.

`first_stub_long`()

adjusts the schedule to have a long stub at the beginning, i.e. first period is longer a regular step.

`last_stub_long`()

adjusts the schedule to have a long stub at the end, i.e. last period is longer a regular step.

class businessdate.businessrange.**BusinessRange**(*start*, *stop=None*, *step=None*, *rolling=None*)

Bases: `list`

class to build list of business days

Parameters

- **start** (BusinessDate) – date to begin schedule, if stop not given, start will be used as stop and default in rolling to `BusinessDate()`
- **stop** (BusinessDate) – date to stop before, if not given, start will be used for stop instead
- **step** (BusinessPeriod) – period to step schedule, if not given 1 day is default
- **rolling** (BusinessDate) – date to roll on (forward and backward) between start and stop, if not given default will be start

Ansatz First, *rolling* and *step* defines a infinite grid of dates. Second, this grid is sliced by *start* (included , if meeting the grid) and *end* (excluded).

All dates will have same **convention**, **holidays** and **day_count** property as **rolling**.

adjust(*convention=None*, *holidays=None*)

returns adjusted *BusinessRange* following given convention

For details of adjusting `BusinessDate` see `BusinessDate.adjust()`.

For possible conventions invoke `BusinessDate().adjust()`

For more details on the conventions see module *conventions*)

3.1.4 BusinessHolidays

class businessdate.businessholidays.**TargetHolidays**(*iterable=()*)

Bases: *businessdate.businessholidays.BusinessHolidays*

holiday calendar class of ecb target2 holidays

Target holidays are

- Jan, 1st
- Good Friday
- Easter Monday
- May, 1st
- December, 25th (Christmas Day)
- December, 25th (Boxing Day)

class businessdate.businessholidays.**BusinessHolidays**(*iterable=()*)

Bases: `list`

holiday calendar class

A *BusinessHolidays* instance imitated a list of `datetime.date` which can be used to check if a `BusinessDate` is included as holiday.

For convenience input need not to be of type `datetime.date`. Duck typing is enough, i.e. having properties *year*, *month* and *day*.

3.2 Convention Functions

3.2.1 Day Count

`businessdate.daycount.`**`diff_in_days`**`(`*start*, *end*`)`
calculates days between start and end date

`businessdate.daycount.`**`get_30_360`**`(`*start*, *end*`)`
implements 30/360 Day Count Convention.

`businessdate.daycount.`**`get_30e_360`**`(`*start*, *end*`)`
implements the 30E/360 Day Count Convention.

`businessdate.daycount.`**`get_30e_360i`**`(`*start*, *end*`)`
implements the 30E/360 I. Day Count Convention.

`businessdate.daycount.`**`get_act_360`**`(`*start*, *end*`)`
implements Act/360 day count convention.

`businessdate.daycount.`**`get_act_365`**`(`*start*, *end*`)`
implements Act/365 day count convention.

`businessdate.daycount.`**`get_act_36525`**`(`*start*, *end*`)`
implements Act/365.25 Day Count Convention

`businessdate.daycount.`**`get_act_act`**`(`*start*, *end*`)`
implements Act/Act day count convention.

3.2.2 Business Day Adjustment

`businessdate.conventions.`**`is_business_day`**`(`*business_date*, *holidays=[]*`)`
method to check if a date falls neither on weekend nor is in holidays.

`businessdate.conventions.`**`adjust_no`**`(`*business_date*, *holidays=()*`)`
does no adjustment.

`businessdate.conventions.`**`adjust_previous`**`(`*business_date*, *holidays=()*`)`
adjusts to Business Day Convention “Preceding”.

`businessdate.conventions.`**`adjust_follow`**`(`*business_date*, *holidays=()*`)`
adjusts to Business Day Convention “Following”.

`businessdate.conventions.`**`adjust_mod_follow`**`(`*business_date*, *holidays=()*`)`
adjusts to Business Day Convention “Modified [Following]”.

`businessdate.conventions.`**`adjust_mod_previous`**`(`*business_date*, *holidays=()*`)`
adjusts to Business Day Convention “Modified Preceding”.

`businessdate.conventions.`**`adjust_start_of_month`**`(`*business_date*, *holidays=()*`)`
adjusts to Business Day Convention “Start of month”, i.e. first business day.

`businessdate.conventions.`**`adjust_end_of_month`**`(`*business_date*, *holidays=()*`)`
adjusts to Business Day Convention “End of month”, i.e. last business day.

`businessdate.conventions.`**`adjust_imm`**`(`*business_date*, *holidays=()*`)`
adjusts to Business Day Convention of “International Monetary Market”.

`businessdate.conventions.`**`adjust_cds_imm`**`(`*business_date*, *holidays=()*`)`
adjusts to Business Day Convention “Single Name CDS”.

CHAPTER

FOUR

RELEASES

These changes are listed in decreasing version number order.

4.1 Release 0.6

Release date was Saturday, 15 January 2022

# moved target_days into BusinessHolidays and removed businessdate.holidays

# added convention, holidays and day_count as BusinessDate arguments as well as properties

# moved to auxilium, development workflow manager

4.2 Release 0.5

Release date was August 1st, 2019

# first beta release (but still work in progress)

# migration to python 3.4, 3.5, 3.6 and 3.7

# automated code review

# 100% test coverage

# finished docs

# removed many calculation functions (BusinessDate.add_years, etc), better use + or - instead

# made some static methods to instance methods (BusinessDate.days_in_month, BusinessDate.end_of_month, BusinessDate.end_of_quarter)

# swapped the order of arguments in *BusinessDate.diff_in_ymd*

# new __cmp__ paradigm

# adding max_days and min_day method to *BusinessPeriod*

4.3 Release 0.4

Release date was December 31th, 2017

4.4 Release 0.3

Release date was July 7th, 2017

4.5 Release 0.2

Release date was April 2nd, 2017

4.6 Release 0.1

Release date was April 1st, 2017

CHAPTER

FIVE

INDICES AND TABLES

- genindex
- modindex
- search

PYTHON MODULE INDEX

INDEX