Spans Documentation

Release 1.1.1

Andreas Runfalk

Apr 21, 2021

Contents

Spans is a pure Python implementation of PostgreSQL's range types. Range types are conveinent when working with intervals of any kind. Every time you've found yourself working with date_start and date_end, an interval may have been what you were actually looking for.

Spans has successfully been used in production since its first release 30th August, 2013.

CHAPTER 1

Example

Imagine you are building a calendar and want to display all weeks that overlaps the current month. Normally you have to do some date trickery to achieve this, since the month's bounds may be any day of the week. With Spans' set-like operations and shortcuts the problem becomes a breeze.

We start by importing `date` and `daterange`

```
>>> from datetime import date
>>> from spans import daterange
```

Using `daterange.from_month` we can get range representing January in the year 2000

```
>>> month = daterange.from_month(2000, 1)
>>> month
daterange(datetime.date(2000, 1, 1), datetime.date(2000, 2, 1))
```

Now we can calculate the ranges for the weeks where the first and last day of month are

```
>>> start_week = daterange.from_date(month.lower, period="week")
>>> end_week = daterange.from_date(month.last, period="week")
>>> start_week
daterange(datetime.date(1999, 12, 27), datetime.date(2000, 1, 3))
>>> end_week
daterange(datetime.date(2000, 1, 31), datetime.date(2000, 2, 7))
```

Using a union we can express the calendar view.

```
>>> start_week.union(month).union(end_week)
daterange(datetime.date(1999, 12, 27), datetime.date(2000, 2, 7))
```

CHAPTER 2

Introduction

2.1 Introduction

For a recent project of mine I started using PostgreSQL's `tsrange` type and needed an equivalent in Python. These range types attempt to mimick PostgreSQL's behavior in every way. To make ranges more useful some extra methods have been added that are not available in PostgreSQL.

2.1.1 Requirements

Spans have no requirements but the standard library. It is known to work on the following Python versions

- Python 2.7
- Python 3.3
- Python 3.4
- Python 3.5
- Python 3.6
- PyPy
- PyPy3

It may work on other version as well.

2.1.2 Installation

Spans is available from PyPI.

```
$ pip install spans
```

2.1.3 Example

If you are making a booking application for a bed and breakfast hotel and want to ensure no room gets double booked:

```python
from collections import defaultdict
from datetime import date
from spans import daterange

# Add a booking from 2013-01-14 through 2013-01-15
bookings = defaultdict(list, {
    1 : [daterange(date(2013, 1, 14), date(2013, 1, 16))]
}

def is_valid_booking(bookings, room, new_booking):
    return not any(booking.overlap(new_booking for booking in bookings[room])

print is_valid_booking(
    bookings, 1, daterange(date(2013, 1, 14), date(2013, 1, 18))) # False
print is_valid_booking(
    bookings, 1, daterange(date(2013, 1, 16), date(2013, 1, 18))) # True
```

2.1.4 Using with Psycopg

To use Spans with Psycopg the Psycospans project exists.

2.2 Ranges

Ranges are like intervals in mathematics. They have a start and end. Every value between the enpoints is included in the range. Integer ranges (*`intrange`*) will be used for all examples. Built in range types are listed in *Available range types*.

A simple range:

```
>>> span = intrange(1, 5)
>>> span.lower
1
>>> span.upper
5
```

By default all ranges include all elements from and including *lower* up to but not including *upper*. This means that the last element included in the discrete *`intrange`* is *4*.

```
>>> intrange(1, 5).last
4
```

Non discrete ranges, such as *`floatrange`*, do not have the property last *`last`*.

Discrete ranges are always normalized, while normal ranges are not.

```
>>> intrange(1, 5, upper_inc=True)
intrange(1, 6)
>>> floatrange(1.0, 5.0, upper_inc=True)
floatrange(1.0, 5.0, upper_inc=True)
```

Ranges support set operations such as *`union`*, *`difference`* and *`intersection`*.

```
>>> intrange(1, 5).union(intrange(5, 10))
intrange(1, 10)
>>> intrange(1, 10).difference(intrange(5, 15))
intrange(1, 5)
>>> intrange(1, 10).intersection(intrange(5, 15))
intrange(5, 10)
```

Unions and differences that would result in two sets will result in a `ValueError`. To perform such operations *Range sets* must be used.

```
>>> intrange(1, 5).union(intrange(10, 15))
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ValueError: Ranges must be either adjacent or overlapping
```

Note: This behavior is for consistency with PostgreSQL.

2.2.1 Available range types

The following range types are built in:

- Integer range (*`intrange`*)
- Float range (*`floatrange`*)
- String range (*`strrange`*) which operate on unicode strings
- Date range (*`daterange`*)
- Datetime range (*`datetimerange`*)
- Timedelta range (*`timedeltarange`*)

2.2.2 Range sets

Range sets are sets of intervals, where each element must be represented by one and only one range. Range sets are the solution to the problem when an operation will result in two separate ranges.

```
>>> intrangeset([intrange(1, 5), intrange(10, 15)])
intrangeset([intrange(1, 5), intrange(10, 15)])
```

Like ranges, range sets support `union`, `difference` and `intersection`. Contrary to Python's built in sets these operations do not modify the range set in place. Instead it returns a new set. Unchanged ranges are reused to conserve memory since ranges are immutable.

Range sets are however mutable structures. To modify an existing set in place the `add` and `remove` methods are used.

```
>>> span = intrangeset([intrange(1, 5)])
>>> span.add(intrange(5, 10))
>>> span
intrangeset([intrange(1, 10)])
>>> span.remove(intrange(3, 7))
>>> span
intrangeset([intrange(1, 3), intrange(7, 10)])
```

2.3 Custom range types

The built in range types may not suffice for your particular application. It is very easy to extend with your own classes. The only requirement is that the type supports rich comparison **PEP 207** and is immutable.

2.3.1 Standard range types

A normal range can be implemented by extending *spans.types.Range*.

```
from spans.types import Range

class floatrange(Range):
        __slots__ = ()
        type = float

span = floatrange(1.0, 5.0)

assert span.lower == 1.0
assert span.upper == 5.0
```

Note: The `__slots__ = ()` is a performance optimization that is used for all ranges. It lowers the memory footprint for every instance. It is not mandatory but encourgaged.

2.3.2 Offsetable range types

An offsetable range can be implemented using the mixin *spans.types.OffsetableRangeMixin*. The class still needs to extend *spans.types.Range*.

```
from spans.types import Range, OffsetableRangeMixin

class floatrange(Range, OffsetableRangeMixin):
        __slots__ = ()
        type = float
```

If the offset type is not the same as the range type (such as `date` that is offsetable with `timedelta`) the attribute `offset_type` can be used.

```
from spans.types import DiscreteRange, OffsetableRangeMixin
from datetime import date, timedelta

class daterange(DiscreteRange, OffsetableRangeMixin):
        __slots__ = ()

        type = date
        offset_type = timedelta

span = daterange(date(2000, 1, 1), date(2000, 2, 1))
assert span.offset(timedelta(14)).upper == date(2000, 2, 15)
```

2.3.3 Discrete range types

Discrete ranges (such as *intrange* and *daterange*) can be implemented by extending *spans.types.DiscreteRange*.

```
from spans.types import DiscreteRange, OffsetableRangeMixin

class intrange(DiscreteRange, OffsetableRangeMixin):
        __slots__ = ()
        type = intrange
        step = 1

assert list(intrange(1, 5)) == [1, 2, 3, 4]
```

Note the `step` attribute. It must always be the smallest possible unit. Using `2` for intranges would not have expected behavior.

2.3.4 Range sets

Range sets are conveinient to implement regardless of the mixins used. This is due to the metaclass *spans.settypes.MetaRangeSet*. The metaclass automatically adds required mixins to the range set type.

```
from spans.types import intrange
from spans.settypes import RangeSet

class intrangeset(RangeSet):
        __slots__ = ()
        type = intrange

assert intrangeset(
        [intrange(1, 5), intrange(10, 15)]).span() == intrange(1, 15)
```

2.3.5 Custom mixins

It is possible to create custom mixins for range sets by adding mappings to *spans.settypes.MetaRangeSet*. The mapping has to be added before the range set class is created or it will not be used.

2.4 Recipes

This is a showcasing of what Spans is capable of.

2.4.1 Check if two employees work at the same time

Spans make working with intervals of time easy. In this example we want to list all hours where *Alice* and *Bob* work at the same time. 24 hour clock is used, as well as weeks starting on Monday.

```
import re
from datetime import timedelta
from spans import timedeltarange, timedeltarangeset
```

(continues on next page)

(continued from previous page)

```python
def str_to_timedeltarange(string):
    """
    Convert a string from the format (HH:MM-HH:MM) into a ``timedeltarange``

    :param string: String time representation in the format HH:MM. Minutes and
                   the leading zero of the hours may be omitted.
    :return: A new ``timedeltarange`` instance
    """

    # NOTE: Error handling left as an exercise for the reader
    match = re.match(
        "^(\d{1,2}):?(\d{1,2})?-(\d{1,2}):?(\d{1,2})?$", string)

    start_hour, start_min = (int(v or 0) for v in match.group(1, 2))
    end_hour, end_min = (int(v or 0) for v in match.group(3, 4))

    return timedeltarange(
        timedelta(hours=start_hour, minutes=start_min),
        timedelta(hours=end_hour, minutes=end_min))


def timedeltarange_to_str(span):
    """
    Convert a ``timedeltarange`` to a string representation (HH:MM-HH:MM).

    :param span: ``timedeltarange`` to convert
    :return: String representation
    """

    return "{:02}:{:02}-{:02}:{:02}".format(
        span.lower.seconds // 3600,
        (span.lower.seconds // 60) % 60,
        span.upper.seconds // 3600,
        (span.upper.seconds // 60) % 60
    )


hours_alice = [
    ["8-12", "12:30-17"], # Monday
    ["8-12", "12:30-17"], # Tuesday
    ["8-12", "12:30-17"], # Wednesday
    ["8-12", "12:30-17"], # Thursday
    ["8-12", "12:30-15"], # Friday
    ["10-14"], # Saturday
    [], # Sunday
]

hours_bob = [
    ["15-21"], # Monday
    ["15-21"], # Tuesday
    ["15-21"], # Wednesday
    ["15-21"], # Thursday
    ["12-18"], # Friday
    [], # Saturday
    ["10-14"], # Sunday
]
```

(continues on next page)

(continued from previous page)

```
schedule_alice = timedeltarangeset(
    str_to_timedeltarange(span).offset(timedelta(day))
    for day, spans in enumerate(hours_alice)
    for span in spans)

schedule_bob = timedeltarangeset(
    str_to_timedeltarange(span).offset(timedelta(day))
    for day, spans in enumerate(hours_bob)
    for span in spans)


# Print hours where both Alice and Bob work
day_names = {
    0: "Monday",
    1: "Tuesday",
    2: "Wednesday",
    3: "Thursday",
    4: "Friday",
    5: "Saturday",
    6: "Sunday",
}
for span in schedule_alice.intersection(schedule_bob):
    print(u"{: <10} {}".format(
        day_names[span.lower.days],
        timedeltarange_to_str(span)))
```

This code outputs:

```
Monday     15:00-17:00
Tuesday    15:00-17:00
Wednesday  15:00-17:00
Thursday   15:00-17:00
Friday     12:30-15:00
```

2.5 PostgreSQL analogies

This page describes range types, functions and operators in PostgreSQL, and what their Spans equivalents are.

2.5.1 Range types

Most range types included in Spans have an equivalent in PostgreSQL.

Postgresql type	Python type
`int4range`	*`intrange`*
`int8range`	*`intrange`*
`numrange`	*`floatrange`*, though *`floatrange`* does not accept integers
`tsrange`	*`datetimerange`*
`tstzrange`	*`datetimerange`*
`daterange`	*`daterange`*
Does not exist[1]	*`timedeltarange`*
Does not exist	*`strrange`*

2.5.2 Operators

Most operators are not overloaded in Python to their PostgreSQL equivalents. Instead Spans implements the functionality using methods.

Operator	PostgreSQL	Python
Equal	`a = b`	`a == b`
Not equal	`a != b` or `a <> b`	`a != b`
Less than	`a < b`	`a < b`
Greater than	`a > b`	`a > b`
Less than or equal	`a < b`	`a < b`
Greater than or equal	`a > b`	`a > b`
Contains	`a @> b`	`a.contains(b)`
Is contained by	`a <@ b`	`a in b` or `a.within(b)`
Overlap	`a && b`	`a.overlap(b)`
Strictly left of	`a << b`	`a.left_of(b)` or `a << b`
Strictly right of	`a >> b`	`a.right_of(b)` or `a >> b`
Does not extend to the right of	`a &< b`	`a.endsbefore(b)`
Does not extend to the left of	`a &> b`	`a.startsafter(b)`
Is adjacent to	`a -|- b`	`a.adjacent(b)`
Union	`a + b`	`a.union(b)` or `a | b`
Intersection	`a * b`	`a.intersection(b)` or `a & b`
Difference	`a - b`	`a.difference(b)` or `a - b`

2.5.3 Functions

There are no functions in Spans that operate on ranges. Instead they are implemented as methods, properties or very simple combinations.

PostgreSQL function	Python equivalent
`lower(a)`	`a.lower`
`upper(a)`	`a.upper`
`isempty(a)`	`a.upper`
`lower_inc(a)`	`a.lower_inc`
`upper_inc(a)`	`a.upper_inc`
`lower_inf(a)`	`a.lower_inf`
`upper_inf(a)`	`a.upper_inf`
`range_merge(a)`	`intrangeset([a, b]).span()`

2.6 API documentation

This is the API reference for all public classes and functions in Spans.

2.6.1 Ranges

[1] Though it is not built in it can be created using: `CREATE TYPE intervalrange AS RANGE(SUBTYPE = interval);`

Range class

class spans.types.**Range**(*lower=None*, *upper=None*, *lower_inc=None*, *upper_inc=None*)

Abstract base class of all ranges.

Ranges are very strict about types. This means that both *lower* or *upper* must be of the given class or subclass or `None`.

All ranges are immutable. No default methods modify the range in place. Instead it returns a new instance.

Parameters

- **lower** – Lower end of range.
- **upper** – Upper end of range.
- **lower_inc** – `True` if lower end should be included in range. Default is `True`
- **upper_inc** – `True` if upper end should be included in range. Default is `False`

Raises

- **TypeError** – If lower or upper bound is not of the correct type.
- **ValueError** – If upper bound is lower than lower bound.

Changed in version 0.5.0: Changed name from `range_` to `Range`

Note: All examples in this class uses *`intrange`* because this class is abstract.

adjacent(*other*)

Returns True if ranges are directly next to each other but does not overlap.

```
>>> intrange(1, 5).adjacent(intrange(5, 10))
True
>>> intrange(1, 5).adjacent(intrange(10, 15))
False
```

The empty set is not adjacent to any set.

This is the same as the `-|-` operator for two ranges in PostgreSQL.

Parameters **other** – Range to test against.

Returns `True` if this range is adjacent with *other*, otherwise `False`.

Raises **TypeError** – If given argument is of invalid type

contains(*other*)

Return True if this contains other. Other may be either range of same type or scalar of same type as the boundaries.

```
>>> intrange(1, 10).contains(intrange(1, 5))
True
>>> intrange(1, 10).contains(intrange(5, 10))
True
>>> intrange(1, 10).contains(intrange(5, 10, upper_inc=True))
False
>>> intrange(1, 10).contains(1)
True
>>> intrange(1, 10).contains(10)
False
```

Contains can also be called using the `in` operator.

```
>>> 1 in intrange(1, 10)
True
```

This is the same as the `self @> other` in PostgreSQL.

Parameters **`other`** – Object to be checked whether it exists within this range or not.

Returns `True` if *other* is completely within this range, otherwise `False`.

Raises **`TypeError`** – If *other* is not of the correct type.

`difference`(*other*)

Compute the difference between this and a given range.

```
>>> intrange(1, 10).difference(intrange(10, 15))
intrange(1, 10)
>>> intrange(1, 10).difference(intrange(5, 10))
intrange(1, 5)
>>> intrange(1, 5).difference(intrange(5, 10))
intrange(1, 5)
>>> intrange(1, 5).difference(intrange(1, 10))
intrange.empty()
```

The difference can not be computed if the resulting range would be split in two separate ranges. This happens when the given range is completely within this range and does not start or end at the same value.

```
>>> intrange(1, 15).difference(intrange(5, 10))
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ValueError: Other range must not be within this range
```

This does not modify the range in place.

This is the same as the `-` operator for two ranges in PostgreSQL.

Parameters **`other`** – Range to difference against.

Returns A new range that is the difference between this and *other*.

Raises **`ValueError`** – If difference bethween this and *other* can not be computed.

`classmethod empty`()

Returns an empty set. An empty set is unbounded and only contain the empty set.

```
>>> intrange.empty() in intrange.empty()
True
```

It is unbounded but the boundaries are not infinite. Its boundaries are returned as `None`. Every set contains the empty set.

`endsbefore`(*other*)

Test if this range ends before *other*. *other* may be either range or scalar. This only takes the upper end of the ranges into consideration. If the scalar or the upper end of the given range is less than or equal to this range's upper end, `True` is returned.

```
>>> intrange(1, 5).endsbefore(5)
True
>>> intrange(1, 5).endsbefore(intrange(1, 5))
True
```

Parameters **other** – Range or scalar to test.

Returns `True` if this range ends before *other*, otherwise `False`

Raises **TypeError** – If *other* is of the wrong type.

endswith(*other*)

Test if this range ends with *other*. *other* may be either range or scalar.

```
>>> intrange(1, 5).endswith(4)
True
>>> intrange(1, 10).endswith(intrange(5, 10))
True
```

Parameters **other** – Range or scalar to test.

Returns `True` if this range ends with *other*, otherwise `False`

Raises **TypeError** – If *other* is of the wrong type.

intersection(*other*)

Returns a new range containing all points shared by both ranges. If no points are shared an empty range is returned.

```
>>> intrange(1, 5).intersection(intrange(1, 10))
intrange(1, 5)
>>> intrange(1, 5).intersection(intrange(5, 10))
intrange.empty()
>>> intrange(1, 10).intersection(intrange(5, 10))
intrange(5, 10)
```

This is the same as the + operator for two ranges in PostgreSQL.

Parameters **other** – Range to interect with.

Returns A new range that is the intersection between this and *other*.

left_of(*other*)

Test if this range *other* is strictly left of *other*.

```
>>> intrange(1, 5).left_of(intrange(5, 10))
True
>>> intrange(1, 10).left_of(intrange(5, 10))
False
```

The bitwise right shift operator << is overloaded for this operation too.

```
>>> intrange(1, 5) << intrange(5, 10)
True
```

The choice of overloading << might seem strange, but it is to mimick PostgreSQL's operators for ranges. As this is not obvious the use of << is discouraged.

Parameters **other** – Range to test against.

Returns `True` if this range is completely to the left of `other`.

overlap(*other*)

Returns True if both ranges share any points.

```
>>> intrange(1, 10).overlap(intrange(5, 15))
True
>>> intrange(1, 5).overlap(intrange(5, 10))
False
```

This is the same as the `&&` operator for two ranges in PostgreSQL.

Parameters **`other`** – Range to test against.

Returns `True` if ranges overlap, otherwise `False`.

Raises **`TypeError`** – If *other* is of another type than this range.

See also:

If you need to know which part that overlapped, consider using *`intersection()`*.

`replace`(*lower=None*, *upper=None*, *lower_inc=None*, *upper_inc=None*)
Returns a new instance of self with the given arguments replaced. It takes the exact same arguments as the constructor.

```
>>> intrange(1, 5).replace(upper=10)
intrange(1, 10)
>>> intrange(1, 10).replace(lower_inc=False)
intrange(2, 10)
>>> intrange(1, 10).replace(5)
intrange(5, 10)
```

Note that range objects are immutable and are never modified in place.

`right_of`(*other*)
Test if this range *other* is strictly right of *other*.

```
>>> intrange(5, 10).right_of(intrange(1, 5))
True
>>> intrange(1, 10).right_of(intrange(1, 5))
False
```

The bitwise right shift operator >> is overloaded for this operation too.

```
>>> intrange(5, 10) >> intrange(1, 5)
True
```

The choice of overloading >> might seem strange, but it is to mimick PostgreSQL's operators for ranges. As this is not obvious the use of >> is discouraged.

Parameters **`other`** – Range to test against.

Returns `True` if this range is completely to the right of `other`.

`startsafter`(*other*)
Test if this range starts after *other*. *other* may be either range or scalar. This only takes the lower end of the ranges into consideration. If the scalar or the lower end of the given range is greater than or equal to this range's lower end, `True` is returned.

```
>>> intrange(1, 5).startsafter(0)
True
>>> intrange(1, 5).startsafter(intrange(0, 5))
True
```

If `other` has the same start as the given

Parameters **other** – Range or scalar to test.

Returns `True` if this range starts after *other*, otherwise `False`

Raises **TypeError** – If *other* is of the wrong type.

startswith(*other*)

Test if this range starts with *other*. *other* may be either range or scalar.

```
>>> intrange(1, 5).startswith(1)
True
>>> intrange(1, 5).startswith(intrange(1, 10))
True
```

Parameters **other** – Range or scalar to test.

Returns `True` if this range starts with *other*, otherwise `False`

Raises **TypeError** – If *other* is of the wrong type.

union(*other*)

Merges this range with a given range.

```
>>> intrange(1, 5).union(intrange(5, 10))
intrange(1, 10)
>>> intrange(1, 10).union(intrange(5, 15))
intrange(1, 15)
```

Two ranges can not be merged if the resulting range would be split in two. This happens when the two sets are neither adjacent nor overlaps.

```
>>> intrange(1, 5).union(intrange(10, 15))
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ValueError: Ranges must be either adjacent or overlapping
```

This does not modify the range in place.

This is the same as the + operator for two ranges in PostgreSQL.

Parameters **other** – Range to merge with.

Returns A new range that is the union of this and *other*.

Raises **ValueError** – If *other* can not be merged with this range.

within(*other*)

Tests if this range is within *other*.

```
>>> a = intrange(1, 10)
>>> b = intrange(3, 8)
>>> a.contains(b)
True
>>> b.within(a)
True
```

This is the same as the `self <@ other` in PostgreSQL. One difference however is that unlike PostgreSQL `self` in this can't be a scalar value.

Parameters **other** – Range to test against.

Returns `True` if this range is completely within the given range, otherwise `False`.

Raises **`TypeError`** – If given range is of the wrong type.

See also:

This method is the inverse of *`contains()`*

`lower`

Returns the lower boundary or None if it is unbounded.

```
>>> intrange(1, 5).lower
1
>>> intrange(upper=5).lower
```

This is the same as the `lower(self)` in PostgreSQL.

`lower_inc`

Returns True if lower bound is included in range. If lower bound is unbounded this returns False.

```
>>> intrange(1, 5).lower_inc
True
```

This is the same as the `lower_inc(self)` in PostgreSQL.

`lower_inf`

Returns True if lower bound is unbounded.

```
>>> intrange(1, 5).lower_inf
False
>>> intrange(upper=5).lower_inf
True
```

This is the same as the `lower_inf(self)` in PostgreSQL.

`upper`

Returns the upper boundary or None if it is unbounded.

```
>>> intrange(1, 5).upper
5
>>> intrange(1).upper
```

This is the same as the `upper(self)` in PostgreSQL.

`upper_inc`

Returns True if upper bound is included in range. If upper bound is unbounded this returns False.

```
>>> intrange(1, 5).upper_inc
False
```

This is the same as the `upper_inc(self)` in PostgreSQL.

`upper_inf`

Returns True if upper bound is unbounded.

```
>>> intrange(1, 5).upper_inf
False
>>> intrange(1).upper_inf
True
```

This is the same as the `upper_inf(self)` in PostgreSQL.

Discrete range

class spans.types.**DiscreteRange**(*lower=None*, *upper=None*, *lower_inc=None*, *upper_inc=None*)

Discrete ranges are a subset of ranges that works on discrete types. This includes `int` and `datetime.date`.

```
>>> intrange(0, 5, lower_inc=False)
intrange(1, 5)
>>> intrange(0, 5, lower_inc=False).lower_inc
True
```

All discrete ranges must provide a unit attribute containing the step length. For intrange this would be:

```
class intrange(DiscreteRange):
    type = int
    unit = 1
```

A range where no values can fit is considered empty:

```
>>> intrange(0, 1, lower_inc=False)
intrange.empty()
```

Discrete ranges are iterable.

```
>>> list(intrange(1, 5))
[1, 2, 3, 4]
```

Changed in version 0.5.0: Changed name from `discreterange` to `DiscreteRange`

endswith(*other*)

Test if this range ends with *other*. *other* may be either range or scalar.

```
>>> intrange(1, 5).endswith(4)
True
>>> intrange(1, 10).endswith(intrange(5, 10))
True
```

Parameters **other** – Range or scalar to test.

Returns `True` if this range ends with *other*, otherwise `False`

Raises **TypeError** – If *other* is of the wrong type.

classmethod next(*curr*)

Increment the given value with the step defined for this class.

```
>>> intrange.next(1)
2
```

Parameters **curr** – Value to increment.

Returns Incremented value.

classmethod prev(*curr*)

Decrement the given value with the step defined for this class.

```
>>> intrange.prev(1)
0
```

Parameters **curr** – Value to decrement.

Returns Decremented value.

last

Returns the last element within this range. If the range has no upper limit `None` is returned.

```
>>> intrange(1, 10).last
9
>>> intrange(1, 10, upper_inc=True).last
10
>>> intrange(1).last is None
True
```

Returns Last element within this range.

New in version 0.1.4.

Offsetable range mixin

class spans.types.**OffsetableRangeMixin**

Mixin for range types that supports being offset by a value. This value must be of the same type as the range boundaries. For date types this will not work and can be solved by explicitly defining an `offset_type`:

```
class datetimerange(Range, OffsetableRangeMixin):
    __slots__ = ()

    type = datetime
    offset_type = timedelta
```

Changed in version 0.5.0: Changed name from `offsetablerange` to `OffsetableRangeMixin`

offset(*offset*)

Shift the range to the left or right with the given offset

```
>>> intrange(0, 5).offset(5)
intrange(5, 10)
>>> intrange(5, 10).offset(-5)
intrange(0, 5)
>>> intrange.empty().offset(5)
intrange.empty()
```

Note that range objects are immutable and are never modified in place.

Parameters **offset** – Scalar to offset by.

New in version 0.1.3.

Integer range

class spans.types.**intrange**(**args*, ***kwargs*)

Range that operates on int.

```
>>> intrange(1, 5)
intrange(1, 5)
```

Inherits methods from *Range*, *DiscreteRange* and *OffsetableRangeMixin*.

Float range

class spans.types.**floatrange**(*lower=None*, *upper=None*, *lower_inc=None*, *upper_inc=None*)

Range that operates on float.

```
>>> floatrange(1.0, 5.0)
floatrange(1.0, 5.0)
>>> floatrange(None, 10.0, upper_inc=True)
floatrange(upper=10.0, upper_inc=True)
```

Inherits methods from `Range` and `OffsetableRangeMixin`.

String range

class spans.types.**strrange**(**args*, ***kwargs*)

Range that operates on unicode strings. Next character is determined lexicographically. Representation might seem odd due to normalization.

```
>>> strrange(u"a", u"z")
strrange(u'a', u'z')
>>> strrange(u"a", u"z", upper_inc=True)
strrange(u'a', u'{')
```

Iteration over a strrange is only sensible when having single character boundaries.

```
>>> list(strrange(u"a", u"e", upper_inc=True))
[u'a', u'b', u'c', u'd', u'e']
>>> len(list(strrange(u"aa", u"zz", upper_inc=True))) # doctest: +SKIP
27852826
```

Inherits methods from `Range` and `DiscreteRange`.

Date range

class spans.types.**daterange**(*lower=None*, *upper=None*, *lower_inc=None*, *upper_inc=None*)

Range that operates on `datetime.date`.

```
>>> daterange(date(2015, 1, 1), date(2015, 2, 1))
daterange(datetime.date(2015, 1, 1), datetime.date(2015, 2, 1))
```

Offsets are done using `datetime.timedelta`.

```
>>> daterange(date(2015, 1, 1), date(2015, 2, 1)).offset(timedelta(14))
daterange(datetime.date(2015, 1, 15), datetime.date(2015, 2, 15))
```

Inherits methods from `Range`, `DiscreteRange` and `OffsetableRangeMixin`.

offset_type

alias of `datetime.timedelta`

type

alias of `datetime.date`

__len__()

Returns number of dates in range.

```
>>> len(daterange(date(2013, 1, 1), date(2013, 1, 8)))
7
```

classmethod from_date(*date*, *period=None*)

Create a day long daterange from for the given date.

```
>>> daterange.from_date(date(2000, 1, 1))
daterange(datetime.date(2000, 1, 1), datetime.date(2000, 1, 2))
```

Parameters

- **date** – A date to convert.
- **period** – The period to normalize date to. A period may be one of: `day` (default), `week`, `american_week`, `month`, `quarter` or `year`.

Returns A new range that contains the given date.

See also:

There are convenience methods for most period types: *from_week()*, *from_month()*, *from_quarter()* and *from_year()*.

PeriodRange has the same interface but is period aware. This means it is possible to get things like next week or month.

Changed in version 0.4.0: Added the period parameter.

classmethod from_month(*year*, *month*)

Create `daterange` based on a year and amonth

Parameters

- **year** – Year as an integer
- **iso_week** – Month as an integer between 1 and 12

Returns A new `daterange` for the given month

New in version 0.4.0.

classmethod from_quarter(*year*, *quarter*)

Create `daterange` based on a year and quarter.

A quarter is considered to be:

- January through March (Q1),
- April through June (Q2),
- July through September (Q3) or,
- October through December (Q4)

Parameters

- **year** – Year as an integer
- **quarter** – Quarter as an integer between 1 and 4

Returns A new `daterange` for the given quarter

New in version 0.4.0.

classmethod from_week(*year*, *iso_week*)

Create `daterange` based on a year and an ISO week

Parameters

- **year** – Year as an integer
- **iso_week** – ISO week number

Returns A new `daterange` for the given week

New in version 0.4.0.

classmethod from_year(*year*)

Create `daterange` based on a year

Parameters **year** – Year as an integer

Returns A new `daterange` for the given year

New in version 0.4.0.

Period range

class spans.types.**PeriodRange**(*lower=None*, *upper=None*, *lower_inc=None*, *upper_inc=None*)

A type aware version of *daterange*.

Type aware refers to being aware of what kind of range it represents. Available types are the same as the `period` argument for to *from_date()*.

Some methods are unavailable due since they don't make sense for *PeriodRange*, and some may return a normal *daterange* since they may modify the range in ways not compatible with its type.

New in version 0.4.0.

Note: This class does not have its own range set implementation, but can be used with *daterangeset*.

difference(*other*)

Compute the difference between this and a given range.

```
>>> intrange(1, 10).difference(intrange(10, 15))
intrange(1, 10)
>>> intrange(1, 10).difference(intrange(5, 10))
intrange(1, 5)
>>> intrange(1, 5).difference(intrange(5, 10))
intrange(1, 5)
>>> intrange(1, 5).difference(intrange(1, 10))
intrange.empty()
```

The difference can not be computed if the resulting range would be split in two separate ranges. This happens when the given range is completely within this range and does not start or end at the same value.

```
>>> intrange(1, 15).difference(intrange(5, 10))
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ValueError: Other range must not be within this range
```

This does not modify the range in place.

This is the same as the − operator for two ranges in PostgreSQL.

Parameters **other** – Range to difference against.

Returns A new range that is the difference between this and *other*.

Raises **ValueError** – If difference bethween this and *other* can not be computed.

classmethod empty()

Raises **TypeError** – since typed date ranges must never be empty

classmethod from_date(*day*, *period=None*)

Create a day long daterange from for the given date.

```
>>> daterange.from_date(date(2000, 1, 1))
daterange(datetime.date(2000, 1, 1), datetime.date(2000, 1, 2))
```

Parameters

- **date** – A date to convert.
- **period** – The period to normalize date to. A period may be one of: `day` (default), `week`, `american_week`, `month`, `quarter` or `year`.

Returns A new range that contains the given date.

See also:

There are convenience methods for most period types: *from_week()*, *from_month()*, *from_quarter()* and *from_year()*.

PeriodRange has the same interface but is period aware. This means it is possible to get things like next week or month.

Changed in version 0.4.0: Added the period parameter.

intersection(*other*)

Returns a new range containing all points shared by both ranges. If no points are shared an empty range is returned.

```
>>> intrange(1, 5).intersection(intrange(1, 10))
intrange(1, 5)
>>> intrange(1, 5).intersection(intrange(5, 10))
intrange.empty()
>>> intrange(1, 10).intersection(intrange(5, 10))
intrange(5, 10)
```

This is the same as the + operator for two ranges in PostgreSQL.

Parameters **other** – Range to interect with.

Returns A new range that is the intersection between this and *other*.

next_period()

The period after this range.

```
>>> span = PeriodRange.from_date(date(2000, 1, 1), period="month")
>>> span.next_period()
PeriodRange(datetime.date(2000, 2, 1), datetime.date(2000, 3, 1), period=
↪'month')
```

Returns A new *PeriodRange* for the period after this period

offset (*offset*)

Offset the date range by the given amount of periods.

This differs from *offset()* on *spans.types.daterange* by not accepting a `timedelta` object. Instead it expects an integer to adjust the typed date range by. The given value may be negative as well.

Parameters **offset** – Number of periods to offset this range by. A period is either a day, week, american week, month, quarter or year, depending on this range's period type.

Returns New offset *PeriodRange*

prev_period ()

The period before this range.

```
>>> span = PeriodRange.from_date(date(2000, 1, 1), period="month")
>>> span.prev_period()
PeriodRange(datetime.date(1999, 12, 1), datetime.date(2000, 1, 1), period=
↪'month')
```

Returns A new *PeriodRange* for the period before this period

replace (*lower=None*, *upper=None*, *lower_inc=None*, *upper_inc=None*)

Returns a new instance of self with the given arguments replaced. It takes the exact same arguments as the constructor.

```
>>> intrange(1, 5).replace(upper=10)
intrange(1, 10)
>>> intrange(1, 10).replace(lower_inc=False)
intrange(2, 10)
>>> intrange(1, 10).replace(5)
intrange(5, 10)
```

Note that range objects are immutable and are never modified in place.

union (*other*)

Merges this range with a given range.

```
>>> intrange(1, 5).union(intrange(5, 10))
intrange(1, 10)
>>> intrange(1, 10).union(intrange(5, 15))
intrange(1, 15)
```

Two ranges can not be merged if the resulting range would be split in two. This happens when the two sets are neither adjacent nor overlaps.

```
>>> intrange(1, 5).union(intrange(10, 15))
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ValueError: Ranges must be either adjacent or overlapping
```

This does not modify the range in place.

This is the same as the + operator for two ranges in PostgreSQL.

Parameters **other** – Range to merge with.

Returns A new range that is the union of this and *other*.

Raises **ValueError** – If *other* can not be merged with this range.

daterange
This `PeriodRange` represented as a naive *`daterange`*.

Datetime range

class `spans.types.`**`datetimerange`**(*lower=None*, *upper=None*, *lower_inc=None*, *upper_inc=None*)

Range that operates on `datetime.datetime`.

```
>>> datetimerange(datetime(2015, 1, 1), datetime(2015, 2, 1))
datetimerange(datetime.datetime(2015, 1, 1, 0, 0), datetime.datetime(2015, 2, 1,␣
↪0, 0))
```

Offsets are done using `datetime.timedelta`.

```
>>> datetimerange(
...     datetime(2015, 1, 1), datetime(2015, 2, 1)).offset(timedelta(14))
datetimerange(datetime.datetime(2015, 1, 15, 0, 0), datetime.datetime(2015, 2, 15,
↪ 0, 0))
```

Inherits methods from *`Range`* and *`OffsetableRangeMixin`*.

Timedelta range

class `spans.types.`**`timedeltarange`**(*lower=None*, *upper=None*, *lower_inc=None*, *upper_inc=None*)

Range that operates on datetime's timedelta class.

```
>>> timedeltarange(timedelta(1), timedelta(5))
timedeltarange(datetime.timedelta(1), datetime.timedelta(5))
```

Offsets are done using `datetime.timedelta`.

```
>>> timedeltarange(timedelta(1), timedelta(5)).offset(timedelta(14))
timedeltarange(datetime.timedelta(15), datetime.timedelta(19))
```

Inherits methods from *`Range`* and *`OffsetableRangeMixin`*.

2.6.2 Range sets

Range set

class `spans.settypes.`**`RangeSet`**(*ranges*)

A range set works a lot like a range with some differences:

- All range sets supports `len()`. Cardinality for a range set means the number of distinct ranges required to represent this set. See *`__len__()`*.
- All range sets are iterable. The iterator returns a range for each iteration. See *`__iter__()`* for more details.
- All range sets are invertible using the ~ operator. The result is a new range set that does not intersect the original range set at all.

```
>>> ~intrangeset([intrange(1, 5)])
intrangeset([intrange(upper=1), intrange(5)])
```

- Contrary to ranges. A range set may be split into multiple ranges when performing set operations such as union, difference or intersection.

Tip: The `RangeSet` constructor supports any iterable sequence as argument.

Parameters **`ranges`** – A sequence of ranges to add to this set.

Raises **`TypeError`** – If any of the given ranges are of incorrect type.

Changed in version 0.5.0: Changed name from `rangeset` to `RangeSet`

`__iter__`()

Returns an iterator over all ranges within this set. Note that this iterates over the normalized version of the range set:

```
>>> list(intrangeset(
...     [intrange(1, 5), intrange(5, 10), intrange(15, 20)]))
[intrange(1, 10), intrange(15, 20)]
```

If the set is empty an empty iterator is returned.

```
>>> list(intrangeset([]))
[]
```

Changed in version 0.3.0: This method used to return an empty range when the RangeSet was empty.

`__len__`()

Returns the cardinality of the set which is 0 for the empty set or else the number of ranges used to represent this range set.

```
>>> len(intrangeset([]))
0
>>> len(intrangeset([intrange(1,5)]))
1
>>> len(intrangeset([intrange(1,5),intrange(10,20)]))
2
```

New in version 0.2.0.

`add`(*item*)

Adds a range to the set.

```
>>> rs = intrangeset([])
>>> rs.add(intrange(1, 10))
>>> rs
intrangeset([intrange(1, 10)])
>>> rs.add(intrange(5, 15))
>>> rs
intrangeset([intrange(1, 15)])
>>> rs.add(intrange(20, 30))
>>> rs
intrangeset([intrange(1, 15), intrange(20, 30)])
```

This operation updates the set in place.

Parameters **`item`** – Range to add to this set.

Raises **`TypeError`** – If any of the given ranges are of incorrect type.

`contains`(*item*)

Test if this range Return True if one range within the set contains elem, which may be either a range of the same type or a scalar of the same type as the ranges within the set.

```
>>> intrangeset([intrange(1, 5)]).contains(3)
True
>>> intrangeset([intrange(1, 5), intrange(10, 20)]).contains(7)
False
>>> intrangeset([intrange(1, 5)]).contains(intrange(2, 3))
True
>>> intrangeset(
...     [intrange(1, 5), intrange(8, 9)]).contains(intrange(4, 6))
False
```

Contains can also be called using the `in` operator.

```
>>> 3 in intrangeset([intrange(1, 5)])
True
```

This operation is *O(n)* where *n* is the number of ranges within this range set.

Parameters **`item`** – Range or scalar to test for.

Returns True if element is contained within this set.

New in version 0.2.0.

`copy`()

Makes a copy of this set. This copy is not deep since ranges are immutable.

```
>>> rs = intrangeset([intrange(1, 5)])
>>> rs_copy = rs.copy()
>>> rs == rs_copy
True
>>> rs is rs_copy
False
```

Returns A new range set with the same ranges as this range set.

`difference`(**others*)

Returns this set stripped of every subset that are in the other given sets.

```
>>> intrangeset([intrange(1, 15)]).difference(
...     intrangeset([intrange(5, 10)]))
intrangeset([intrange(1, 5), intrange(10, 15)])
```

Parameters **`other`** – Range set to compute difference against.

Returns A new range set that is the difference between this and *other*.

`intersection`(**others*)

Returns a new set of all subsets that exist in this and every given set.

```
>>> intrangeset([intrange(1, 15)]).intersection(
...     intrangeset([intrange(5, 10)]))
intrangeset([intrange(5, 10)])
```

Parameters **`other`** – Range set to intersect this range set with.

Returns A new range set that is the intersection between this and *other*.

`remove`(*item*)

Remove a range from the set. This operation updates the set in place.

```
>>> rs = intrangeset([intrange(1, 15)])
>>> rs.remove(intrange(5, 10))
>>> rs
intrangeset([intrange(1, 5), intrange(10, 15)])
```

Parameters **`item`** – Range to remove from this set.

`span`()

Return a range that spans from the first point to the last point in this set. This means the smallest range containing all elements of this set with no gaps.

```
>>> intrangeset([intrange(1, 5), intrange(30, 40)]).span()
intrange(1, 40)
```

This method can be used to implement the PostgreSQL function `range_merge(a, b)`:

```
>>> a = intrange(1, 5)
>>> b = intrange(10, 15)
>>> intrangeset([a, b]).span()
intrange(1, 15)
```

Returns A new range the contains this entire range set.

`union`(**others*)

Returns this set combined with every given set into a super set for each given set.

```
>>> intrangeset([intrange(1, 5)]).union(
...     intrangeset([intrange(5, 10)]))
intrangeset([intrange(1, 10)])
```

Parameters **`other`** – Range set to merge with.

Returns A new range set that is the union of this and *other*.

Discrete range set mixin

class `spans.settypes.`**`DiscreteRangeSetMixin`**

Mixin that adds support for discrete range set operations. Automatically used by *`RangeSet`* when *`Range`* type inherits *`DiscreteRange`*.

Changed in version 0.5.0: Changed name from `discreterangeset` to `DiscreteRangeSetMixin`

`values`()

Returns an iterator over each value in this range set.

```
>>> list(intrangeset([intrange(1, 5), intrange(10, 15)]).values())
[1, 2, 3, 4, 10, 11, 12, 13, 14]
```

Offsetable range set mixin

class spans.settypes.**OffsetableRangeSetMixin**

Mixin that adds support for offsetable range set operations. Automatically used by *RangeSet* when range type inherits `OffsetableRangeMixin`.

Changed in version 0.5.0: Changed name from `offsetablerangeset` to `OffsetableRangeSetMixin`

offset(*offset*)

Shift the range set to the left or right with the given offset

```
>>> intrangeset([intrange(0, 5), intrange(10, 15)]).offset(5)
intrangeset([intrange(5, 10), intrange(15, 20)])
>>> intrangeset([intrange(5, 10), intrange(15, 20)]).offset(-5)
intrangeset([intrange(0, 5), intrange(10, 15)])
```

This function returns an offset copy of the original set, i.e. updating is not done in place.

Integer range set

class spans.settypes.**intrangeset**(*ranges*)

Range set that operates on *intrange*.

```
>>> intrangeset([intrange(1, 5), intrange(10, 15)])
intrangeset([intrange(1, 5), intrange(10, 15)])
```

Inherits methods from *RangeSet*, `DiscreteRangeset` and `OffsetableRangeMixinset`.

Float range set

class spans.settypes.**floatrangeset**(*ranges*)

Range set that operates on *floatrange*.

```
>>> floatrangeset([floatrange(1.0, 5.0), floatrange(10.0, 15.0)])
floatrangeset([floatrange(1.0, 5.0), floatrange(10.0, 15.0)])
```

Inherits methods from *RangeSet*, `DiscreteRangeset` and `OffsetableRangeMixinset`.

String range set

class spans.settypes.**strrangeset**(*ranges*)

Range set that operates on .. seealso:: *strrange*.

```
>>> strrangeset([
...     strrange(u"a", u"f", upper_inc=True),
...     strrange(u"0", u"9", upper_inc=True)])
strrangeset([strrange(u'0', u':'), strrange(u'a', u'g')])
```

Inherits methods from *RangeSet* and `DiscreteRangeset`.

Date range set

class spans.settypes.**daterangeset**(*ranges*)

Range set that operates on *daterange*.

```
>>> month = daterange(date(2000, 1, 1), date(2000, 2, 1))
>>> daterangeset([month, month.offset(timedelta(366))]) # doctest: +NORMALIZE_
↪WHITESPACE
daterangeset([daterange(datetime.date(2000, 1, 1), datetime.date(2000, 2, 1)),
    daterange(datetime.date(2001, 1, 1), datetime.date(2001, 2, 1))])
```

Inherits methods from *RangeSet*, DiscreteRangeset and OffsetableRangeMixinset.

Datetime range set

class spans.settypes.**datetimerangeset**(*ranges*)

Range set that operates on *datetimerange*.

```
>>> month = datetimerange(datetime(2000, 1, 1), datetime(2000, 2, 1))
>>> datetimerangeset([month, month.offset(timedelta(366))]) # doctest: +NORMALIZE_
↪WHITESPACE
datetimerangeset([datetimerange(datetime.datetime(2000, 1, 1, 0, 0), datetime.
↪datetime(2000, 2, 1, 0, 0)),
    datetimerange(datetime.datetime(2001, 1, 1, 0, 0), datetime.datetime(2001, 2,␣
↪1, 0, 0))])
```

Inherits methods from *RangeSet* and OffsetableRangeMixinset.

Timedelta range set

class spans.settypes.**timedeltarangeset**(*ranges*)

Range set that operates on *timedeltarange*.

```
>>> week = timedeltarange(timedelta(0), timedelta(7))
>>> timedeltarangeset([week, week.offset(timedelta(7))])
timedeltarangeset([timedeltarange(datetime.timedelta(0), datetime.timedelta(14))])
```

Inherits methods from *RangeSet* and OffsetableRangeMixinset.

Meta range set

class spans.settypes.**MetaRangeSet**

A meta class for RangeSets. The purpose is to automatically add relevant mixins to the range set class based on what mixins and base classes the range class has.

All subclasses of *RangeSet* uses this class as its metaclass

Changed in version 0.5.0: Changed name from metarangeset to MetaRangeSet

2.6.3 Legacy names

Historically some internal Spans classes had all lowercase names. This was changed in version 0.5.0. The reason some classes still have lowercase names is to match the Python built-ins they map to. date's range type is and will always be *daterange*. However, it doesn't make much sense to maintain this convention for the more hidden classes in Spans.

`spans.types.`**`range_`**

This alias exist for legacy reasons. It is considered deprecated but will not likely be removed.

New in version 0.5.0.

alias of *spans.types.Range*

`spans.types.`**`discreterange`**

This alias exist for legacy reasons. It is considered deprecated but will not likely be removed.

New in version 0.5.0.

alias of *spans.types.DiscreteRange*

`spans.types.`**`offsetablerange`**

This alias exist for legacy reasons. It is considered deprecated but will not likely be removed.

New in version 0.5.0.

alias of *spans.types.OffsetableRangeMixin*

`spans.settypes.`**`metarangeset`**

This alias exist for legacy reasons. It is considered deprecated but will not likely be removed.

New in version 0.5.0.

alias of *spans.settypes.MetaRangeSet*

`spans.settypes.`**`rangeset`**

This alias exist for legacy reasons. It is considered deprecated but will not likely be removed.

New in version 0.5.0.

alias of *spans.settypes.RangeSet*

2.7 Changelog

Version are structured like the following: `<major>.<minor>.<bugfix>`. The first *0.1* release does not properly adhere to this. Unless explicitly stated, changes are made by Andreas Runfalk.

2.7.1 Version 1.1.1

Released on 21st April, 2021

- Normalize ranges to be empty when start and end is the same and either bound is exclusive (bug #18, lgharib-ashvili)

2.7.2 Version 1.1.0

Released on 2nd June, 2019

This release changes a lot of internal implementation details that should prevent methods from not handling unbounded ranges correctly in the future.

- Added validation to ensure unbounded ranges are never inclusive

- Changed `__repr__` for ranges to be more similar to proper Python syntax. The old representation looked like mismatched parentheses. For instance `floatrange((,10.0])` becomes `floatrange(upper=10.0, upper_inc=True)`
- Dropped Python 3.3 support since it's been EOL for almost two years. It probably still works but it is no longer tested
- Fixed pickling of empty range sets not working (bug #14)
- Fixed `union()` not working properly with unbounded ranges
- Fixed lowerly unbounded ranges improperly being lower inclusive
- Fixed `startswith()` and `endsbefore()` being not handling empty ranges

2.7.3 Version 1.0.2

Released on 22th February, 2019

- Fixed `union()` when `upper_inc` is set to `True` (bug #11, Michael Krahe)

2.7.4 Version 1.0.1

Released on 31st January, 2018

- Fixed `PartialOrderingMixin` not using `__slots__` (bug #10)

2.7.5 Version 1.0.0

Released on 8th June, 2017

- Added `NotImplemented` for `<<` and `>>` operators when there is a type mismatch
- Added `|` operator for unions of `Range` and `NotImplemented` support for `RangeSet`
- Added `&` operator for intersections of `Range` and `NotImplemented` support for `RangeSet`
- Added `-` operator for differences of `Range` and `NotImplemented` support for `RangeSet`
- Added `reversed()` iterator support for `DiscreteRange`
- Fixed overlap with empty range incorrectly returns `True` (bug #7)
- Fixed issue with `contains()` for scalars on unbounded ranges
- Fixed type check for `right_of()`
- Fixed type check for `contains()`
- Fixed type check for `union()`
- Fixed type check for `intersection()`
- Fixed type check for `difference()`
- Fixed infinite iterators not being supported for `DiscreteRange`

2.7.6 Version 0.5.0

Released on 16th April, 2017

This release is a preparation for a stable 1.0 release.

- Fixed comparison operators when working with empty or unbounded ranges. They would previously raise exceptions. Ranges are now partially ordered instead of totally ordered
- Added more unit tests
- Renamed classes to match **PEP 8#class-names** conventions. This does not apply to classes that works on built-in that does not follow **PEP 8#class-names**.
- Refactored `left_of()`
- Refactored `overlap()`
- Refactored `union()`

2.7.7 Version 0.4.0

Released on 20th March, 2017

This release is called 0.4.1 on PyPI because I messed up the upload.

- Added new argument to `from_date()` for working with different kinds of date intervals. The argument accepts a period of either `"day"` (default), `"week"` (ISO week), `"american_week"` (starts on sunday), `"month"`, `"quarter"` or `"year"`.
- Added new methods to `daterange` for working with different kinds of date intervals: `from_week()`, `from_month()`, `from_quarter()` and `from_year()`.
- Added a new class `PeriodRange` for working with periods like weeks, months, quarters or years. It inherits all methods from `daterange` and is aware of its own period type. It allows things like getting the previous or next week.
- Fixed `daterange` not accepting subclasses of `date` (bug #5)
- Fixed some broken doctests
- Moved unit tests to pytest
- Removed Tox config
- Minor documentation tweaks

2.7.8 Version 0.3.0

Released on 26th August, 2016

- Added documentation for `__iter__()`
- Fixed intersection of multiple range sets not working correctly (bug #3)
- Fixed iteration of `RangeSet` returning an empty range when `RangeSet` is empty (bug #4)

Warning: This change is backwards incompatible to code that expect range sets to always return at least one set when iterating.

2.7.9 Version 0.2.1

Released on 27th June, 2016

- Fixed `RangeSet` not returning `NotImplemented` when comparing to classes that are not sub classes of `RangeSet`, pull request #2 (Michael Krahe)
- Updated license in `setup.py` to follow recommendations by PyPA

2.7.10 Version 0.2.0

Released on 22nd December, 2015

- Added `__len__()` to range sets (Michael Krahe)
- Added `contains()` to range sets (Michael Krahe)
- Added Sphinx style doc strings to all methods
- Added proper Sphinx documentation
- Added unit tests for uncovered parts, mostly error checking
- Added wheel to PyPI along with source distribution
- Fixed a potential bug where comparing ranges of different types would result in an infinite loop
- Changed meta class implementation for range sets to allow more mixins for custom range sets

2.7.11 Version 0.1.4

Released on 15th May, 2015

- Added `last` property to `DiscreteRange`
- Added `from_date()` helper to `daterange`
- Added more unit tests
- Improved pickle implementation
- Made type checking more strict for date ranges to prevent `datetime` from being allowed in `daterange`

2.7.12 Version 0.1.3

Released on 27th February, 2015

- Added `offset()` to some range types
- Added `offset()` to some range set types
- Added sanity checks to range boundaries
- Fixed incorrect `__slots__` usage, resulting in `__slots__` not being used on most ranges
- Fixed pickling of ranges and range sets
- Simplified creation of new range sets, by the use of the meta class `MetaRangeSet`

2.7.13 Version 0.1.2

Released on 13th June, 2014

- Fix for inproper version detection on Ubuntu's bundled Python interpreter

2.7.14 Version 0.1.1

Released on 12th June, 2014

- Readme fixes
- Syntax highlighting for PyPI page

2.7.15 Version 0.1.0

Released on 30th August, 2013

- Initial release

Python Module Index

Index