multirange Documentation

Release 0.3.0

ibu radempa

May 10, 2015

1 Status 3

2 Introduction 5
2.1 Overview . . . 5
2.2 Features . . . 5
2.3 Limitations . . . 5
2.4 Range . . . 5
2.5 Generalized range object . . . 6
2.6 Range iterable . . . 6
2.7 Multirange . . . 6

3 Usage examples 7

4 Functions 9

Python Module Index 13

multiranges provides functions operating on multiple range-like objects.

Convenience functions for multiple range-like objects

An elementary package for Python >= 3.3

https://pypi.python.org/pypi/multirange/

CHAPTER 1

Status

The code works, but it is not stable: functionality might be added or reorganized as long as the major version equals 0 (cf. http://semver.org/spec/v2.0.0.html, item #4). Hint: Stability grows quicker when you provide feedback.

multirange is not yet feature complete; most operations involving multiranges are missing.

CHAPTER 2

Introduction

2.1 Overview

This library for Python >= 3.3 provides convenience functions for multiple range-like objects corresponding to finite sets of consecutive integers.

It has 3 main types of operations:

- operations involving few range-like objects (a generalization of Python's native range objects)
- operations involving an iterable of range-like objects (*range iterables*)
- operations involving so-called multiranges; we define a *multirange* as iterables range-like objects, which have no mutual overlap, which are not adjacent, and which are ordered increasingly.

2.2 Features

- Provide operations on multiple instances of *range* (disregarding attribute *step*), or any other object having attributes *start* and *stop* evaluating to `int`

  Note: Since Python 3.3 `range` objects have the *start*, *stop* and *step* attributes.

- Avoid materializing of ranges as full lists of integers. Instead, results are computed from the boundaries (start, stop) only.
- If not otherwise noted, the functions of this module throw no Exceptions, provided they are called with valid parameters.

2.3 Limitations

- Require Python >= 3.3

2.4 Range

In the context of this module we define as a *range r* either a native Python `range` object, or any other object having attributes *start* and *stop*, which evaluate to `int`.

A range *r* has the meaning of the set of all consecutive integers from r.start to r.stop - 1. If r.start >= r.stop, this means the empty set. Note that for negative step values the native Python `range` object may generate several values, while in our context an empty set may result. Example: range(0, -10, -1) generates 10 values, while in our context (step == 1) this entails an empty set of integers.

Ranges often need to be brought to normal form (cf. *normalize()*). By default the normal form is a native `range` object with step == 1, or None if r.stop <= r.start. Alternatively, in case r.stop > r.start, the normal form may be any other *generalized range object*, which is obtained using a non-default value of the *construct* keyword argument in most functions (see below).

The functions of this module always accept ranges in their normalized form, and if not otherwise stated, non-normalized ranges are accepted, too.

Two ranges are called *adjacent* if the end (value of the stop attribute) of one coincides with the beginning (value of the start attribute) of the other.

2.5 Generalized range object

When the documentation of this module refers to a *range*, it usually means a *generalized range object* (or *range-like object*), not just Python's native `range`.

As *generalized range object* we define an object which can be constructed using exactly two integer arguments, *start* and *stop*, and which has attributes *start* and *stop* returning these integer values at any time. One example is the native `range` object. Here is another very simple one:

```
class MyRange(object):
    def __init__(self, start, stop):
        self.start = start
        self.stop = stop
```

The main advantage of generalized range objects over native range objects is that they may have additional structure beyond *start* and *stop* (where the native range only has a *step* attribute).

2.6 Range iterable

The purpose of this module is to ease common operations involving multiple ranges, more precisely, iterables of ranges. By *range iterable* we mean an iterable yielding either None or an instance of a generalized range object.

> **Warning:** Some functions need to sort range iterables, thereby defining an intermediate list, so don't expect optimal performance for iterables with a large number of items for all functions.

Range iterables are not to be confused with multiranges.

2.7 Multirange

As *multirange* we define a range iterable where the ranges don't overlap, are not adjacent and are ordered increasingly. A *multirange* can be obtained from any *range iterable* by using *normalize_multi()*.

CHAPTER 3

Usage examples

```
>>> import multirange as mr
>>> print(mr.normalize(range(5, 0)))
None
>>> mr.overlap(range(0, 10), range(5, 15))
range(5, 10)
>>> mr.is_disjunct([range(8, 10), range(0, 2), range(2, 4)])
True
>>> mr.covering_all([range(8, 10), range(0, 2), range(2, 4)])
range(0, 10)
>>> mr.contains(range(0, 10), range(0, 5))
True
>>> mr.is_covered_by([range(8, 10), range(0, 2)], range(0, 20))
True
>>> mr.intermediate(range(10, 15), range(0, 5))
range(5, 10)
>>> list(mr.gaps([range(4, 6), range(6, 7), range(8, 10), range(0, 3)]))
[range(3, 4), range(7, 8)]
>>> mr.difference(range(1, 9), range(2, 3))
(range(1, 2), range(3, 9))
>>> list(mr.normalize_multi([None, range(0, 5), range(5, 7), range(8, 9)]))
[range(0, 7), range(8, 9)]
>>> list(mr.difference_one_multi(range(0, 9), [range(-2, 2), range(4, 5)]))
[range(2, 4), range(5, 9)]
```

Please consult the unit tests (latest) for more examples.

CHAPTER 4

Functions

multirange.**normalize**(*r*, *construct=<class ‘range’>*)

Return an object which is the normalization of range *r*.

The normalized range is either None (if r.start >= r.stop), or an object constructed using *construct* with the arguments r.start, r.stop.

In case construct == range we try to avoid constructing new objects.

multirange.**filter_normalize**(*rs*, *construct=<class ‘range’>*)

Normalize ranges iteratively.

Iterate over all ranges in the given range iterable *rs*, yielding normalized ranges

multirange.**filter_nonempty**(*rs*, *invert=False*, *do_normalize=True*, *construct=<class ‘range’>*, *with_position=False*)

Filter for non-empty ranges.

Iterate over all ranges in the given range iterable *rs* and yield those which are not None after normalization; if *invert* is True, yield those which are None

If *do_normalize* is True, yield only normalized non-empty ranges (using the constructor given in *construct* upon normalization); otherwise yield the original range objects.

If with_position is True, return 2-tuples consisting of the position of the matching range within *rs* and the matching range. Otherwise yield only the matching range.

multirange.**equals**(*r1*, *r2*)

Check equality of two ranges.

Return whether the the two ranges *r1* and *r2* are equal after normalization.

Incidental remark: If you have native range objects (being not None) and want to take into account step values, you can use native python equality of ranges; for instance, range(0, 5, -10) == range(0, -5) == range(0).

multirange.**filter_equal**(*rs*, *r*, *do_normalize=True*, *construct=<class ‘range’>*, *with_position=False*)

Filter ranges for equality to a given range.

Iterate over all ranges in the given range iterable *rs* and yield those which are equal to range *r* after normalization.

If *do_normalize* evaluates to True, then do not return the original items from *rs*, but instead normalized ranges, where the range objects are constructed using *construct*.

If *with_position* evalues to True, then yield 2-tuples consisting of an `int` indicating the position of a matching range within *rs* and the range itself.

`multirange.`**`is_adjacent`**(*r1*, *r2*)

Check for adjacency of two ranges.

Return whether the ranges *r1* and *r2* are adjacent.

If *r1* or *r2* is None after normalization, return None instead of a `bool`.

`multirange.`**`overlap`**(*r1*, *r2*, *construct=<class 'range'>*)

Overlap of two ranges.

For two ranges *r1* and *r2* return the normalized range corresponding to the intersection ot the sets (of consecutive integers) corresponding to *r1* and *r2*

Return a normalized result, which is either None, or an object constructed using *construct*.

`multirange.`**`filter_overlap`**(*rs*, *r*, *do_normalize=False*, *construct=<class 'range'>*, *with_position=False*)

Filter for ranges overlapping with a given range.

Iterate over the range iterable *rs*, and yield only those ranges having a non-vanishing overlap with range *r*.

Note: Some of the original ranges are yielded, not their overlapping parts.

If *do_normalize* evaluates to True, then do not return the original items from *rs*, but instead normalized range objects constructed using *construct*.

If *with_position* evalues to True, then yield 2-tuples consisting of an `int` indicating the position of a matching range within *rs* and the range itself.

`multirange.`**`match_count`**(*rs*, *r*)

Count matches with a gievn range.

Return the number of ranges yielded from iterable *rs*, which have a non-vanishing overlap with range *r*.

`multirange.`**`overlap_all`**(*rs*, *construct=<class 'range'>*)

Overlap of all given ranges.

Return the range corresponding to the intersection of the sets of integers corresponding to the ranges obtained from the iterable *rs*

Return a normalized result, where the normalized object is constructed using *construct*.

`multirange.`**`is_disjunct`**(*rs*, *assume_ordered_increasingly=False*)

Check for disjointness of all given ranges.

Return whether the range iterable *rs* consists of mutually disjunct ranges.

If *assume_ordered_increasingly* is True, only direct neighbors (qua iteration order) are checked for non-vanishing overlap.

`multirange.`**`covering_all`**(*rs*, *construct=<class 'range'>*)

Return the smallest covering range for the ranges in range iterable *rs*.

Return a normalized result, where the normalized object is constructed using *construct*.

`multirange.`**`contains`**(*r1*, *r2*)

Check inclusion of two ranges.

Return whether range *r1* contains range *r2*.

`multirange.`**`filter_contained`**(*rs*, *r*, *do_normalize=False*, *construct=<class 'range'>*, *with_position=False*)

Filter for ranges contained in a given range.

Yield those ranges from range iterable *rs*, which are contained in range *r*.

If *do_normalize* evaluates to True, then do not return the original items from *rs*, but instead normalized range objects constructed using *construct*.

If *with_position* evalues to True, then yield 2-tuples consisting of an `int` indicating the position of a matching range within *rs* and the range itself.

`multirange.`**`is_covered_by`**`(`*rs*, *r*`)`
Check inclusion of ranges in a given range.

Return whether range *r* covers all ranges from range iterable *rs*.

`multirange.`**`symmetric_difference`**`(`*r1*, *r2*, *construct=<class 'range'>*`)`
Symmetric difference of two ranges.

Return the symmetric difference between range *r1* and range *r2* as two range-like objects (constructed using *construct*, and possibly None), where the first corresponds to a subset or *r1* and the second corresponds to a subset or *r2*

Instead of ranges, *r1* and *r2* can also be range-like objects.

Note: The resulting range-like objects correspond to disjunct sets of integers, but they need not be ordered, if *r1* and *r2* are not.

`multirange.`**`intermediate`**`(`*r1*, *r2*, *construct=<class 'range'>*, *assume_ordered=False*`)`
Intermediate of two ranges.

Return the range inbetween range *r1* and range *r2*, or None if they overlap or if at least one of them corresponds to an empty set.

Return a normalized range object constructed using *construct*.

`multirange.`**`sort_by_start`**`(`*rs*`)`
Sorted list of ranges.

Return a list of (unmodified) ranges obtained from range iterable *rs*, sorted by their start values, and omitting empty ranges.

`multirange.`**`gaps`**`(`*rs*, *construct=<class 'range'>*, *assume_ordered=False*`)`
Find gaps between ranges.

Yield the gaps between the ranges from range iterable *rs*, i.e., the maximal ranges without overlap with any of the ranges, but within the covering range.

Yield normalized, non-empty range objects constructed using *construct*.

`multirange.`**`is_partition_of`**`(`*rs*, *construct=<class 'range'>*, *assume_ordered=False*`)`
Check if ranges are a partition.

Return the covering range of the ranges from range iterable *rs*, if they have no gaps; else return None.

The covering range is constructed using *construct*.

`multirange.`**`difference`**`(`*r1*, *r2*, *construct=<class 'range'>*`)`
Difference of two ranges.

Return two ranges resulting when the integers from range *r2* are removed from range *r1*.

Return two ranges: the first being the part below *r2* and the second the one above *r2*. They may both be None. In the special case where *r2* after normalization equals None, return (r1, None) (i.e., take the difference to be the lower part).

The range-like objects are constructed using *construct*.

`multirange.`**`normalize_multi`**`(`*rs*, *construct=<class 'range'>*, *assume_ordered_increasingly=False*`)`
Return a *normalized* multirange from the given range iterable *rs*.

Overlapping or adjacent ranges are merged into one, and the ranges are ordered increasingly.

Yield normalized ranges. Don't yield None.

multirange.**difference_one_multi**(*r*, *mr*, *construct=<class 'range'>*)

Subtract multirange *mr* from range *r*, resulting in a multirange.

The range-like objects generated by this function are constructed using *construct*.

multirange.**multi_intersection**(*mr1*, *mr2*, *construct=<class 'range'>*)

Intersection of two multiranges.

Return a multirange consisting of range-like objects which are intersections of the ranges in multirange *mr1* and multirange *mr2*.

More precisely, the resulting multirange corresponds to the set of integers which is the intersection of the sets of integers corresponding to *mr1* and *mr2*.

The range-like objects generated by this function are constructed using *construct*. (Note: They are newly constructed, even if items from *mr1* or *mr2* have the required values for the *start* and *stop* attributes.)

multirange.**multi_union**(*mr1*, *mr2*, *construct=<class 'range'>*)

Union of two multiranges.

Return a multirange consisting of range-like objects which are unions of the ranges in multirange *mr1* and multirange *mr2*

More precisely, the resulting multirange corresponds to the set of integers which is the union of the sets of integers corresponding to *mr1* and *mr2*.

The range-like objects generated by this function are constructed using *construct*. (Note: They are newly constructed, even if items from *mr1* or *mr2* have the required values for the *start* and *stop* attributes.)

Python Module Index

m

multirange, 1

C

contains() (in module multirange), 10
covering_all() (in module multirange), 10

D

difference() (in module multirange), 11
difference_one_multi() (in module multirange), 12

E

equals() (in module multirange), 9

F

filter_contained() (in module multirange), 10
filter_equal() (in module multirange), 9
filter_nonempty() (in module multirange), 9
filter_normalize() (in module multirange), 9
filter_overlap() (in module multirange), 10

G

gaps() (in module multirange), 11

I

intermediate() (in module multirange), 11
is_adjacent() (in module multirange), 9
is_covered_by() (in module multirange), 11
is_disjunct() (in module multirange), 10
is_partition_of() (in module multirange), 11

M

match_count() (in module multirange), 10
multi_intersection() (in module multirange), 12
multi_union() (in module multirange), 12
multirange (module), 1

N

normalize() (in module multirange), 9
normalize_multi() (in module multirange), 11

O

overlap() (in module multirange), 10
overlap_all() (in module multirange), 10

S

sort_by_start() (in module multirange), 11
symmetric_difference() (in module multirange), 11