ivalutils Documentation

Release 0.8.0

Michael Amrhein

Sep 04, 2017

Contents

The package *ivalutils* provides classes for basic interval arithmetics as well as classes for building sequences of adjacent intervals and for building mappings of intervals to arbitrary values.

Contents:

CHAPTER 1

Interval

Basic interval arithmetic.

Usage

Creating intervals

The class *Interval* is used to create intervals, i. e. subsets of a set of values, by defining a lower and an upper endpoint.

The simplest way is calling *Interval* without arguments, resulting in both endpoints to be infinite:

```
>>> ival = Interval()
>>> ival
Interval()
>>> str(ival)
'(-inf .. +inf)'
```

For getting a more useful interval, it's neccessary to specify atleast one endpoint:

```
>>> ival = Interval(LowerClosedLimit(0))
>>> ival
Interval(lower_limit=Limit(True, 0, True))
>>> str(ival)
'[0 .. +inf)'
```

```
>>> ival = Interval(upper_limit=UpperClosedLimit(100.))
>>> ival
Interval(upper_limit=Limit(False, 100.0, True))
>>> str(ival)
'(-inf .. 100.0]'
```

```
>>> ival = Interval(LowerClosedLimit(0), UpperOpenLimit(27))
>>> ival
Interval(lower_limit=Limit(True, 0, True), upper_limit=Limit(False, 27, False))
>>> str(ival)
'[0 .. 27)'
```

Any type which defines a total ordering can be used for the limits:

```
>>> ClosedInterval('a', 'zzz')
Interval(lower_limit=Limit(True, 'a', True), upper_limit=Limit(False, 'zzz', True))
```

Several factory functions can be used as shortcut. For example:

```
>>> LowerClosedInterval(30)
Interval(lower_limit=Limit(True, 30, True))
>>> UpperOpenInterval(0)
Interval(upper_limit=Limit(False, 0, False))
>>> ClosedInterval(1, 3)
Interval(lower_limit=Limit(True, 1, True), upper_limit=Limit(False, 3, True))
>>> ChainableInterval(0, 5)
Interval(lower_limit=Limit(True, 0, True), upper_limit=Limit(False, 5, False))
```

Operations on intervals

The limits of an interval can be retrieved via properties:

```
>>> ival = ClosedInterval(0, 100)
>>> ival.lower_limit
Limit(True, 0, True)
>>> ival.upper_limit
Limit(False, 100, True)
>>> ival.limits
(Limit(True, 0, True), Limit(False, 100, True))
```

Several methods can be used to test for specifics of an interval. For example:

```
>>> ival.is_bounded()
True
>>> ival.is_finite()
True
>>> ival.is_left_open()
False
```

Intervals can be tested for including a value:

```
>>> 74 in ival
True
>>> -4 in ival
False
```

Intervals can be compared:

```
>>> ival2 = LowerOpenInterval(100)
>>> ival3 = LowerClosedInterval(100)
>>> ival < ival2
True
```

```
>>> ival < ival3
True
>>> ival2 < ival3
False
>>> ival2 == ival3
False
>>> ival3 < ival2
True
>>> ival2.is_adjacent(ival3)
False
>>> ival3.is_adjacent(ival2)
False
>>> ival4 = UpperClosedInterval(100)
>>> ival4.is_adjacent(ival2)
True
>>> ival.is_overlapping(ival3)
True
>>> ival.is_subset(ival4)
True
```

Classes

class ivalutils.interval.**Limit**(*lower*, *value*, *closed=True*)

Lower / upper limit of an Interval.

Parameters

- **lower** (*bool*) – specifies which endpoint of an interval this limit defines: *True* -> lower endpoint, *False* -> upper endpoint
- **value** (*<total ordering>*) – limiting value (can be of any type that defines a total ordering)
- **closed** (*bool*) – specifies whether value itself is the endpoint or not: *True* -> the interval that has this limit includes value, *False* -> the interval that has this limit does not includes value

Returns instance of *Limit*

Raises

- AssertionError – *lower* is not instance of *bool*
- AssertionError – *value* is None
- AssertionError – *closed* is not instance of *bool*

__eq__(*other*)

self == other

__ge__(*other*)

self >= other

__gt__(*other*)

self > other

__hash__()

hash(self)

__le__(*other*)
self <= other

__lt__(*other*)
self < other

adjacent_limit()
Return the limit adjacent to self.

class ivalutils.interval.**Interval**(*lower_limit=None*, *upper_limit=None*)
An Interval defines a subset of a set of values by optionally giving a lower and / or an upper limit.

The base set of values - and therefore the given limits - must have a common base type which defines a total order on the values.

If both limits are given, the interval is said to be *bounded* or *finite*, if only one or neither of them is given, the interval is said to be *unbounded* or *infinite*.

If only the lower limit is given, the interval is called *lower bounded* or *left bounded* (maybe also *upper unbounded*, *upper infinite*, *right unbounded* or *right infinite*). Correspondingly, if only the upper limit is given, the interval is called *upper bounded* or *right bounded* (maybe also *lower unbounded*, *lower infinite*, *left unbounded* or *left infinite*).

For both limits (aka endpoints) must be specified whether the given value is included in the interval or not. In the first case the limit is called *closed*, otherwise *open*.

Parameters

- **lower_limit** (*Limit*) – lower limit (default: None)
- **upper_limit** (*Limit*) – upper limit (default: None)

Returns instance of *Interval*

If *None* is given as *lower_limit*, the resulting interval is lower infinite. If *None* is given as *upper_limit*, the resulting interval is upper infinite.

Raises

- *InvalidInterval* – *lower_limit* is not a lower limit
- *InvalidInterval* – *upper_limit* is not a upper limit
- *InvalidInterval* – *lower_limit* > *upper_limit*
- *IncompatibleLimits* – values of *lower_limit* and *upper_limit* are not comparable

__and__(*other*)
self & other

__contains__(*value*)
True if value does not exceed the limits of self.

__copy__()
Return self (Interval instances are immutable).

__eq__(*other*)
self == other.

True if all elements contained in self are also contained in other and all elements contained in other are also contained in self.

This is exactly the case if self.limits == other.limits.

__ge__(*other*)
self >= other.

__gt__(*other*)
self > other.

True if there is an element in self which is greater than all elements in other or there is an element in other which is smaller than all elements in self.

This is exactly the case if self.limits > other.limits.

__hash__()
hash(self)

__le__(*other*)
self <= other.

__lt__(*other*)
self < other.

True if there is an element in self which is smaller than all elements in other or there is an element in other which is greater than all elements in self.

This is exactly the case if self.limits < other.limits.

__or__(*other*)
self | other

__sub__(*other*)
self - other

is_adjacent(*other*)
True if self.is_lower_adjacent(other) or self.is_upper_adjacent(other).

is_disjoint(*other*)
True if self contains no elements in common with other.

is_lower_adjacent(*other*)
True if self.upper_limit.is_lower_adjacent(other.lower_limit).

is_overlapping(*other*)
True if there is a common element in self and other.

is_subset(*other*)
True if self defines a proper subset of other, i.e. all elements contained in self are also contained in other, but not the other way round.

is_upper_adjacent(*other*)
True if self.lower_limit.is_upper_adjacent(other.upper_limit).

limits
Lower and upper limit as tuple.

lower_limit
Lower limit (LowerInfiniteLimit, if no lower limit was given.)

upper_limit
Upper limit (UpperInfiniteLimit, if no upper limit was given.)

Factoryfunctions

ivalutils.interval.**LowerLimit**(*value*, *closed=True*)
Create a lower limit.

Parameters

- **value** (*<total ordering>*) – limiting value (can be of any type that defines a total ordering)
- **closed** (*bool*) – specifies whether value itself is the endpoint or not: *True* -> the interval that has this limit includes value, *False* -> the interval that has this limit does not includes value

Returns instance of *Limit*

Raises

- AssertionError – *value* is None
- AssertionError – *closed* is not instance of *bool*

ivalutils.interval.**LowerInfiniteLimit**()
Create a lower infinite limit (a singleton).

ivalutils.interval.**UpperLimit**(*value*, *closed=True*)
Create an upper limit.

Parameters

- **value** (*<total ordering>*) – limiting value (can be of any type that defines a total ordering)
- **closed** (*bool*) – specifies whether value itself is the endpoint or not: *True* -> the interval that has this limit includes value, *False* -> the interval that has this limit does not includes value

Returns instance of *Limit*

Raises

- AssertionError – *value* is None
- AssertionError – *closed* is not instance of *bool*

ivalutils.interval.**UpperInfiniteLimit**()
Create an upper infinite limit (a singleton).

ivalutils.interval.**LowerClosedLimit**(*value*)
Create a lower closed limit.

Parameters **value** (*<total ordering>*) – limiting value (can be of any type that defines a total ordering)

Returns instance of *Limit*

Raises AssertionError – *value* is None

ivalutils.interval.**LowerOpenLimit**(*value*)
Create a lower open limit.

Parameters **value** (*<total ordering>*) – limiting value (can be of any type that defines a total ordering)

Returns instance of *Limit*

Raises AssertionError – *value* is None

ivalutils.interval.**UpperClosedLimit**(*value*)
Create an upper closed limit.

Parameters **value** (*<total ordering>*) – limiting value (can be of any type that defines a total ordering)

Returns instance of *Limit*

Raises AssertionError – *value* is None

ivalutils.interval.**UpperOpenLimit**(*value*)
Create an upper open limit.

Parameters **value** (*<total ordering>*) – limiting value (can be of any type that defines a total ordering)

Returns instance of *Limit*

Raises AssertionError – *value* is None

ivalutils.interval.**ChainableInterval**(*lower_value*, *upper_value*, *lower_closed=True*)
Create Interval with one closed and one open endpoint.

ivalutils.interval.**ClosedInterval**(*lower_value*, *upper_value*)
Create Interval with closed endpoints.

ivalutils.interval.**LowerClosedInterval**(*lower_value*)
Create Interval with closed lower and infinite upper endpoint.

ivalutils.interval.**LowerOpenInterval**(*lower_value*)
Create Interval with open lower and infinite upper endpoint.

ivalutils.interval.**OpenBoundedInterval**(*lower_value*, *upper_value*)
Create Interval with open endpoints.

ivalutils.interval.**OpenFiniteInterval**(*lower_value*, *upper_value*)
Create Interval with open endpoints.

ivalutils.interval.**UpperClosedInterval**(*upper_value*)
Create Interval with infinite lower and closed upper endpoint.

ivalutils.interval.**UpperOpenInterval**(*upper_value*)
Create Interval with infinite lower and open upper endpoint.

Exceptions

class ivalutils.interval.**IncompatibleLimits**
Raised when comparing limits with incompatible types of values.

class ivalutils.interval.**InvalidInterval**
Raised when an invalid Interval would be created.

CHAPTER 2

IntervalChain

Sequences of adjacent intervals.

Usage

Creating interval chains

The class *IntervalChain* is used to create sequences of adjacent intervals:

```
>>> ic = IntervalChain(('a', 'd', 'g', 'z'))
>>> ic
IntervalChain(('a', 'd', 'g', 'z'))
```

The default is to create an interval sequence which is lower-bound and upper-infinite and containing lower-closed intervals:

```
>>> str(ic)
"[['a' .. 'd'), ['d' .. 'g'), ['g' .. 'z'), ['z' .. +inf)]"
```

By specifying additional parameters, you can determine which endpoints will be closed and whether a lower and / or upper infinite endpoint will be added:

```
>>> ic = IntervalChain(('a', 'd', 'g', 'z'), lower_closed = False, add_lower_inf=True,
↪ add_upper_inf=False)
>>> str(ic)
"[(-inf .. 'a'], ('a' .. 'd'], ('d' .. 'g'], ('g' .. 'z']]"
```

Operations on interval chains

Interval chains can be indexed and iterated like lists ...:

```
>>> ic[2]
Interval(lower_limit=Limit(True, 'd', False), upper_limit=Limit(False, 'g', True))
>>> [ival.upper_limit.value for ival in ic]
['a', 'd', 'g', 'z']
```

... and can be searched for the index of the interval holding a specified value:

```
>>> ic.map2idx('b')
1
>>> ic.map2idx('a')
0
>>> ic.map2idx('aa')
1
```

Classes

class ivalutils.interval_chain.**IntervalChain**(*limits*, *lower_closed=True*, *add_lower_inf=False*, *add_upper_inf=True*)

An IntervalChain is a list of adjacent intervals.

It is constructed from a list of limiting values.

Parameters

- **limits** (*Iterable*) – an iterable of the limiting values, must be ordered from smallest to greatest
- **lower_closed** (*boolean*) – defines which endpoint of the contained intervals will be closed: if *True*, lower endpoint closed, upper open (default), if *False*, lower endpoint open, upper closed
- **add_lower_inf** (*boolean*) – defines whether a lower infinite interval will be added as first interval: if *True*, infinite interval as lowest interval, if *False*, no infinite interval as lowest interval (default)
- **add_upper_inf** (*boolean*) – defines whether an upper infinite interval will be added as last interval: if *True*, infinite interval as last interval (default), if *False*, no infinite interval as last interval

Returns instance of *IntervalChain*

Raises

- *EmptyIntervalChain* – given limits do not define any interval
- InvalidInterval – given limits do not define a sequence of adjacent intervals
- IncompatibleLimits – given limits are not comparable

__copy__()
Return self (IntervalChain instances are immutable).

__eq__(*other*)
self == other

__getitem__(*idx*)
self[idx]

__iter__()
iter(self)

__len__()
len(self)

__repr__()
repr(self)

__str__()
str(self)

is_lower_infinite()
True if first interval is lower infinite.

is_upper_infinite()
True if last interval is upper infinite.

map2idx(*value*)
Return the index of the interval which contains value.

Raises ValueError if value is not contained in any of the intervals in self.

limits
The limiting values.

total_interval
Returns the interval between lower endpoint of first interval in self and upper endpoint of last interval in self.

Exceptions

class ivalutils.interval_chain.**EmptyIntervalChain**
Raised when an empty IntervalChain would be created.

CHAPTER 3

IntervalMapping

Mappings on intervals

Usage

Creating interval mappings

The class *IntervalMapping* is used to create a mapping from intervals to arbitrary values.

Instances can be created by giving an IntervalChain and a sequence of associated values ...:

```
>>> im1 = IntervalMapping(IntervalChain((0, 300, 500, 1000)), (0., .10, .15, .20))
```

... or a sequence of limiting values and a sequence of associated values ...:

```
>>> im2 = IntervalMapping((0, 300, 500, 1000), (0., .10, .15, .20))
```

... or a sequence of tuples, each holding a limiting value and an associated value:

```
>>> im3 = IntervalMapping(((0, 0.), (300, .10), (500, .15), (1000, .20)))
>>> im1 == im2 == im3
True
```

Operations on IntervalMappings

Interval mappings behave like ordinary mappings:

```
>>> list(im3.keys())
[Interval(lower_limit=Limit(True, 0, True), upper_limit=Limit(False, 300, False)),
 Interval(lower_limit=Limit(True, 300, True), upper_limit=Limit(False, 500, False)),
 Interval(lower_limit=Limit(True, 500, True), upper_limit=Limit(False, 1000, False)),
 Interval(lower_limit=Limit(True, 1000, True))]
```

```
>>> list(im3.values())
[0.0, 0.1, 0.15, 0.2]
>>> im3[Interval(lower_limit=Limit(True, 300, True), upper_limit=Limit(False, 500,␣
↪False))]
0.1
```

In addition they can be looked-up for the value associated with the interval which contains a given value:

```
>>> im3.map(583)
0.15
```

As a short-cut, the interval mapping can be used like a function:

```
>>> im3(412)
0.1
```

Use cases for interval mappings are for example:

- determine the discount to be applied depending on an order value,
- rating customers depending on their sales turnover,
- classifying cities based on the number of inhabitants,
- mapping booking dates to accounting periods,
- grouping of measured values in discrete ranges.

Classes

class ivalutils.interval_map.**IntervalMapping**(**args*)

An IntervalMapping is a container of associated interval / value pairs.

It is constructed from either

- an IntervalChain and a sequence of associated values,
- a sequence of limiting values and a sequence of associated values,
- a sequence of tuples, each holding a limiting value and an associated value.

1. Form

Parameters

- **arg0** (IntervalChain) – sequence of intervals to be mapped
- **arg1** (*Sequence*) – sequence of associated values

2. Form

Parameters

- **arg0** (*Sequence*) – sequence of values limiting the intervals to be mapped
- **arg1** (*Sequence*) – sequence of associated values

3. Form

Parameters **arg0** (*Sequence*) – sequence of tuples containing a limiting value and an associated value

If no IntervalChain is given, the given limiting values must be comparable and must be given in ascending order.

Returns instance of *IntervalMapping*

Raises

- `AssertionError` – given sequences do not have the same length
- `AssertionError` – given sequences of limiting values is empty
- `InvalidInterval` – given limits do not define a sequence of adjacent intervals
- `IncompatibleLimits` – given limits are not comparable
- `TypeError` – given sequence is not a sequence of 2-tuples
- `TypeError` – wrong number of arguments

__call__(*val*)
Return the value associated with interval which contains *val*.

__copy__()
Return self (IntervalMapping instances are immutable).

__eq__(*other*)
self == other

__getitem__(*key*)
self[key]

__iter__()
iter(self)

__len__()
len(self)

map(*val*)
Return the value associated with interval which contains *val*.

genindex

Python Module Index

i

Index

Symbols

A

C

E

I

L

M

O

T

U