Package ‘set6’

October 18, 2021

Title R6 Mathematical Sets Interface
Version 0.2.4
Description An object-oriented package for mathematical sets, upgrading the current gold-standard {sets}. Many forms of mathematical sets are implemented, including (countably finite) sets, tuples, intervals (countably infinite or uncountable), and fuzzy variants. Wrappers extend functionality by allowing symbolic representations of complex operations on sets, including unions, (cartesian) products, exponentiation, and differences (asymmetric and symmetric).

LinkingTo Rcpp
Imports checkmate, ooplah, Rcpp, R6, utils
Suggests knitr, testthat, devtools, rmarkdown
License MIT + file LICENSE
BugReports https://github.com/xoopR/set6/issues
VignetteBuilder knitr
Encoding UTF-8
RoxygenNote 7.1.1
SystemRequirements C++11
Collate 'Properties.R' 'RcppExports.R' 'Set.R' 'SetWrapper.R'
'SetWrapper_ComplementSet.R' 'SetWrapper_ExponentSet.R'
'SetWrapper_PowersetSet.R' 'SetWrapper_ProductSet.R'
'SetWrapper_UnionSet.R' 'Set_Complex.R' 'Set_ConditionalSet.R'
'Set_FuzzySet.R' 'Set_FuzzySet_FuzzyMultiset.R'
'Set_FuzzySet_FuzzyTuple.R' 'Set_Interval.R' 'setSymbol.R'
'Set_Interval_SpecialSet.R' 'Set_LogicalSet.R' 'Set_Logicals.R'
'Set_Multiset.R' 'Set_Tuple.R' 'Set_Universal.R'
'Set_UniversalSet.R' 'asFuzzySet.R' 'asInterval.R' 'asSet.R'
'helpers.R' 'assertions.R' 'atomic_coercions.R'
'listSpecialSets.R' 'operation_cleaner.R'
'operation_powerset.R' 'operation_setcomplement.R'
'operation_setintersect.R' 'operation_setpower.R'
'operation_setproduct.R' 'operation_setsymdiff.R'
'operation_setunion.R' 'operators.R' 'set6-deprecated.R'
'set6-package.R' 'set6.news.R' 'useUnicode.R' 'zzz.R'
NeedsCompilation: yes
Author: Raphael Sonabend [aut, cre] (<https://orcid.org/0000-0001-9225-4654>),
    Franz Kiraly [aut]
Maintainer: Raphael Sonabend <raphaelsonabend@gmail.com>
Repository: CRAN
Date/Publication: 2021-10-18 14:20:02 UTC

R topics documented:

set6-package .......................................................... 3
as.FuzzySet ............................................................ 4
as.Interval ............................................................. 7
as.Set ................................................................. 8
ComplementSet ......................................................... 10
Complex ............................................................... 12
ConditionalSet ......................................................... 15
contains ............................................................... 19
equals ................................................................. 19
ExponentSet ........................................................... 20
ExtendedReals .......................................................... 21
FuzzyMultiset .......................................................... 22
FuzzySet ............................................................... 25
FuzzyTuple ............................................................. 30
Integers ................................................................. 33
Interval ................................................................. 34
isSubset ............................................................... 39
listSpecialSets ....................................................... 40
Logicals ............................................................... 40
LogicalSet ............................................................. 41
Multiset ............................................................... 42
Naturals ............................................................... 44
NegIntegers ............................................................ 45
NegRationals ........................................................... 46
NegReals ............................................................... 47
PosIntegers ............................................................ 48
PosNaturals ............................................................ 49
PosRationals ........................................................... 50
PosReals ............................................................... 51
powerset .............................................................. 52
PowersetSet ............................................................ 53
ProductSet ............................................................. 55
Properties ............................................................. 57
Rationals .............................................................. 58
Reals ................................................................. 59
Set ................................................................. 60
set6News ............................................................. 68
### Description

`set6` upgrades the `{sets}` package to R6. Many forms of mathematical sets are implemented, including (countably finite) sets, tuples, intervals (countably infinite or uncountable), and fuzzy variants. Wrappers extend functionality by allowing symbolic representations of complex operations on sets, including unions, (cartesian) products, exponentiation, and differences (asymmetric and symmetric).
Details

The main features of set6 are:

- Object-oriented programming, which allows a clear inheritance structure for Sets, Intervals, Tuples, and other variants.
- Set operations and wrappers for both explicit and symbolic representations for algebra of sets.
- Methods for assertions and comparison checks, including subsets, equality, and containedness.

To learn more about set6, start with the set6 vignette:

vignette("set6","set6")

And for more advanced usage see the complete tutorials at

https://github.com/xoopR/set6

Author(s)

Maintainer: Raphael Sonabend <raphaelsonabend@gmail.com> (ORCID)

Authors:

- Franz Kiraly <f.kiraly@ucl.ac.uk>

See Also

Useful links:

- https://xoopR.github.io/set6/
- https://github.com/xoopR/set6

as.FuzzySet

Coercion to R6 FuzzySet/FuzzyTuple

Description

Coerces object to an R6 FuzzySet/FuzzyTuple

Usage

as.FuzzySet(object)

## S3 method for class 'numeric'
as.FuzzySet(object)

## S3 method for class 'list'
as.FuzzySet(object)
as.FuzzySet

## S3 method for class 'matrix'
as.FuzzySet(object)

## S3 method for class 'data.frame'
as.FuzzySet(object)

## S3 method for class 'Set'
as.FuzzySet(object)

## S3 method for class 'FuzzySet'
as.FuzzySet(object)

## S3 method for class 'Interval'
as.FuzzySet(object)

## S3 method for class 'ConditionalSet'
as.FuzzySet(object)

as.FuzzyTuple(object)

## S3 method for class 'numeric'
as.FuzzyTuple(object)

## S3 method for class 'list'
as.FuzzyTuple(object)

## S3 method for class 'matrix'
as.FuzzyTuple(object)

## S3 method for class 'data.frame'
as.FuzzyTuple(object)

## S3 method for class 'Set'
as.FuzzyTuple(object)

## S3 method for class 'FuzzySet'
as.FuzzyTuple(object)

## S3 method for class 'Interval'
as.FuzzyTuple(object)

## S3 method for class 'ConditionalSet'
as.FuzzyTuple(object)

as.FuzzyMultiset(object)

## S3 method for class 'numeric'
as.FuzzyMultiset(object)
## S3 method for class 'list'
as.FuzzyMultiset(object)

## S3 method for class 'matrix'
as.FuzzyMultiset(object)

## S3 method for class 'data.frame'
as.FuzzyMultiset(object)

## S3 method for class 'Set'
as.FuzzyMultiset(object)

## S3 method for class 'FuzzySet'
as.FuzzyMultiset(object)

## S3 method for class 'Interval'
as.FuzzyMultiset(object)

## S3 method for class 'ConditionalSet'
as.FuzzyMultiset(object)

Arguments

object object to coerce

Details

- as.FuzzySet.list - Assumes list has two items, named elements and membership, and that they are ordered to be corresponding.
- as.FuzzySet.matrix - Assumes first column corresponds to elements and second column corresponds to their respective membership.
- as.FuzzySet.data.frame - First checks to see if one column is called elements and the other is called membership. If not then uses as.FuzzySet.matrix.
- as.FuzzySet.Set - Creates a FuzzySet by assuming Set elements all have membership equal to 1.
- as.FuzzySet.Interval - First tries coercion via as.Set.Interval then uses as.FuzzySet.Set.

See Also

FuzzySet FuzzyTuple

Other coercions: as.Interval(), as.Set()
Coercion to R6 Interval

Description
Coerces object to an R6 Interval.

Usage

```r
as.Interval(object)
```

## S3 method for class 'Set'
```r
as.Interval(object)
```

## S3 method for class 'Interval'
```r
as.Interval(object)
```

## S3 method for class 'list'
```r
as.Interval(object)
```

## S3 method for class 'data.frame'
```r
as.Interval(object)
```

## S3 method for class 'matrix'
```r
as.Interval(object)
```

## S3 method for class 'numeric'
```r
as.Interval(object)
```

## S3 method for class 'ConditionalSet'
```r
as.Interval(object)
```

Arguments

- **object** object to coerce

Details

- `as.Interval.list/as.Interval.data.frame` - Assumes the list/data.frame has named items/columns: lower, upper, type, class.
- `as.Interval.numeric` - If the numeric vector is a continuous interval with no breaks then coerces to an `Interval` with: lower = min(object), upper = max(object), class = "integer". Ordering is ignored.
- `as.Interval.matrix` - Tries coercion via `as.Interval.numeric` on the first column of the matrix.
- `as.Interval.Set` - First tries coercion via `as.Interval.numeric`, if possible wraps result in a `Set`.
• `as.Interval.FuzzySet` - Tries coercion via `as.Interval.Set` on the support of the `FuzzySet`.

See Also

`Interval`

Other coercions: `as.FuzzySet()`, `as.Set()`

---

### as.Set

**Coercion to R6 Set/Tuple**

**Description**

Coerces object to an R6 `Set/Tuple`

**Usage**

```r
as.Set(object)
```

```r
## Default S3 method:
as.Set(object)
```

```r
## S3 method for class 'numeric'
as.Set(object)
```

```r
## S3 method for class 'list'
as.Set(object)
```

```r
## S3 method for class 'matrix'
as.Set(object)
```

```r
## S3 method for class 'data.frame'
as.Set(object)
```

```r
## S3 method for class 'Set'
as.Set(object)
```

```r
## S3 method for class 'FuzzySet'
as.Set(object)
```

```r
## S3 method for class 'Interval'
as.Set(object)
```

```r
## S3 method for class 'ConditionalSet'
as.Set(object)
```

```r
as.Tuple(object)
```
## Default S3 method:
as.Tuple(object)

## S3 method for class 'numeric'
as.Tuple(object)

## S3 method for class 'list'
as.Tuple(object)

## S3 method for class 'matrix'
as.Tuple(object)

## S3 method for class 'data.frame'
as.Tuple(object)

## S3 method for class 'FuzzySet'
as.Tuple(object)

## S3 method for class 'Set'
as.Tuple(object)

## S3 method for class 'Interval'
as.Tuple(object)

## S3 method for class 'ConditionalSet'
as.Tuple(object)

as.Multiset(object)

## Default S3 method:
as.Multiset(object)

## S3 method for class 'numeric'
as.Multiset(object)

## S3 method for class 'list'
as.Multiset(object)

## S3 method for class 'matrix'
as.Multiset(object)

## S3 method for class 'data.frame'
as.Multiset(object)

## S3 method for class 'FuzzySet'
as.Multiset(object)

## S3 method for class 'Set'
as.Multiset(object)
## S3 method for class 'Interval'
as.Multiset(object)
## S3 method for class 'ConditionalSet'
as.Multiset(object)

**Arguments**

- object: object to coerce

**Details**

- as.Set.default: Creates a Set using the object as the elements.
- as.Set.list: Creates a Set for each element in list.
- as.Set.matrix/as.Set.data.frame: Creates a Set for each column in matrix/data.frame.
- as.Set.FuzzySet: Creates a Set from the support of the FuzzySet.
- as.Set.Interval: If the interval has finite cardinality then creates a Set from the Interval elements.

**See Also**

- Set Tuple

Other coercions: as.FuzzySet(), as.Interval()

---

**ComplementSet**

*Set of Complements*

**Description**

ComplementSet class for symbolic complement of mathematical sets.

**Details**

The purpose of this class is to provide a symbolic representation for the complement of sets that cannot be represented in a simpler class. Whilst this is not an abstract class, it is not recommended to construct this class directly but via the set operation methods.

**Super classes**

set6::Set -> set6::SetWrapper -> ComplementSet


**Active bindings**

- `elements` Returns the elements in the object.
- `length` Returns the number of elements in the object.
- `addedSet` For the ComplementSet wrapper, X-Y, returns the set X.
- `subtractedSet` For the ComplementSet wrapper, X-Y, returns the set Y.

**Methods**

**Public methods:**

- `ComplementSet$new()`
- `ComplementSet$strprint()`
- `ComplementSet$contains()`
- `ComplementSet$clone()`

**Method** `new()`: Create a new ComplementSet object. It is not recommended to construct this class directly.

*Usage:*

```r
ComplementSet$new(addset, subtractset, lower = NULL, upper = NULL, type = NULL)
```

*Arguments:*

- `addset` Set to be subtracted from.
- `subtractset` Set to subtract.
- `lower` lower bound of new object.
- `upper` upper bound of new object.
- `type` closure type of new object.

*Returns:* A new ComplementSet object.

**Method** `strprint()`: Creates a printable representation of the object.

*Usage:*

```r
ComplementSet$strprint(n = 2)
```

*Arguments:*

- `n` numeric. Number of elements to display on either side of ellipsis when printing.

*Returns:* A character string representing the object.

**Method** `contains()`: Tests if elements `x` are contained in `self`.

*Usage:*

```r
ComplementSet[contains(x, all = FALSE, bound = FALSE)]
```

*Arguments:*

- `x` Set or vector of Sets.
- `all` logical. If FALSE tests each `x` separately. Otherwise returns TRUE only if all `x` pass test.
- `bound` logical
Returns: If all == TRUE then returns TRUE if all x are contained in self, otherwise FALSE. If all == FALSE returns a vector of logicals corresponding to the length of x, representing if each is contained in self. If bound == TRUE then an element is contained in self if it is on or within the (possibly-open) bounds of self, otherwise TRUE only if the element is within self or the bounds are closed.

Method clone(): The objects of this class are cloneable with this method.

Usage:
ComplementSet$clone(deep = FALSE)

Arguments:
deep Whether to make a deep clone.

See Also
Set operations: setunion, setproduct, setpower, setcomplement, setsymdiff, powerset, setintersect
Other wrappers: ExponentSet, PowersetSet, ProductSet, UnionSet

<table>
<thead>
<tr>
<th>Complex</th>
<th>Set of Complex Numbers</th>
</tr>
</thead>
</table>

Description
The mathematical set of complex numbers, defined as the the set of reals with possibly imaginary components. i.e.

\[ a + bi : a, b \in R \]

where \( R \) is the set of reals.

Details
There is no inherent ordering in the set of complex numbers, hence only the contains method is implemented here.

Super class
set6::Set -> Complex

Methods

Public methods:
- Complex$new()
- Complex$contains()
- Complex>equals()
- Complex$isSubset()
- Complex$strprint()
- Complex$clone()
Method new(): Create a new Complex object.

Usage:
Complex$new()

Returns: A new Complex object.

Method contains(): Tests to see if x is contained in the Set.

Usage:
Complex$contains(x, all = FALSE, bound = NULL)

Arguments:
x any. Object or vector of objects to test.
all logical. If FALSE tests each x separately. Otherwise returns TRUE only if all x pass test.
bound logical.

Details: x can be of any type, including a Set itself. x should be a tuple if checking to see if it lies within a set of dimension greater than one. To test for multiple x at the same time, then provide these as a list.
If all = TRUE then returns TRUE if all x are contained in the Set, otherwise returns a vector of logicals. For Intervals, bound is used to specify if elements lying on the (possibly open) boundary of the interval are considered contained (bound = TRUE) or not (bound = FALSE).

Returns: If all is TRUE then returns TRUE if all elements of x are contained in the Set, otherwise FALSE. If all is FALSE then returns a vector of logicals corresponding to each individual element of x.
The infix operator %inset% is available to test if x is an element in the Set, see examples.

Method equals(): Tests if two sets are equal.

Usage:
Complex$equals(x, all = FALSE)

Arguments:
x Set or vector of Sets.
all logical. If FALSE tests each x separately. Otherwise returns TRUE only if all x pass test.

Returns: If all is TRUE then returns TRUE if all x are equal to the Set, otherwise FALSE. If all is FALSE then returns a vector of logicals corresponding to each individual element of x.
Infix operators can be used for:

Equal ==
Not equal !=

Examples:

# Equals
Set$new(1,2)$equals(Set$new(5,6))
Set$new(1,2)$equals(Interval$new(1,2))
Set$new(1,2) == Interval$new(1,2, class = "integer")

# Not equal
\!Set\new(1,2) = Set\new(1,2)\\
Set\new(1,2) \neq Set\new(1,5)

**Method** `isSubset()`: Test if one set is a (proper) subset of another

*Usage:*
Complex\$isSubset(x, proper = FALSE, all = FALSE)

*Arguments:*
x any. Object or vector of objects to test.
propre logical. If TRUE tests for proper subsets.
all logical. If FALSE tests each x separately. Otherwise returns TRUE only if all x pass test.

*Details:* If using the method directly, and not via one of the operators then the additional boolean argument `proper` can be used to specify testing of subsets or proper subsets. A Set is a proper subset of another if it is fully contained by the other Set (i.e. not equal to) whereas a Set is a (non-proper) subset if it is fully contained by, or equal to, the other Set.

When calling `isSubset` on objects inheriting from `Interval`, the method treats the interval as if it is a `Set`, i.e. ordering and class are ignored. Use `$isSubinterval` to test if one interval is a subinterval of another.

Infix operators can be used for:

- Subset `<`
- Proper Subset `<=`
- Superset `>`
- Proper Superset `>=`

Every `Set` is a subset of a `Universal`. No `Set` is a super set of a `Universal`, and only a `Universal` is not a proper subset of a `Universal`.

*Returns:* If all is TRUE then returns TRUE if all x are subsets of the Set, otherwise FALSE. If all is FALSE then returns a vector of logicals corresponding to each individual element of x.

*Examples:*
Set\$new(1,2,3)$isSubset(Set\$new(1,2), proper = TRUE)
Set\$new(1,2) < Set\$new(1,2,3) # proper subset

\(c\)\(\{\text{Set}\$new(1,2,3), Set\$new(1)}\) < Set\$new(1,2,3) # not proper
Set\$new(1,2,3) <= Set\$new(1,2,3) # proper

**Method** `strprint()`: Creates a printable representation of the object.

*Usage:*
Complex\$strprint(n = 2)

*Arguments:*
n numeric. Number of elements to display on either side of ellipsis when printing.

*Returns:* A character string representing the object.

**Method** `clone()`: The objects of this class are cloneable with this method.

*Usage:*
ConditionalSet

Complex$clone(deep = FALSE)

Arguments:

deep  Whether to make a deep clone.

See Also

Other special sets: ExtendedReals, Integers, Logicals, Naturals, NegIntegers, NegRationals, NegReals, PosIntegers, PosNaturals, PosRationals, PosReals, Rationals, Reals, Universal

Examples

## ------------------------------------------------
## Method `Complex$equals`
## ------------------------------------------------

# Equals
Set$new(1,2)$equals(Set$new(5,6))
Set$new(1,2)$equals(Interval$new(1,2))
Set$new(1,2) == Interval$new(1,2, class = "integer")

# Not equal
!Set$new(1,2)$equals(Set$new(1,2))
Set$new(1,2) != Set$new(1,5)

## ------------------------------------------------
## Method `Complex$isSubset`
## ------------------------------------------------

Set$new(1,2,3)$isSubset(Set$new(1,2), proper = TRUE)
Set$new(1,2) < Set$new(1,2,3) # proper subset
c(Set$new(1,2,3), Set$new(1)) < Set$new(1,2,3) # not proper
Set$new(1,2,3) <= Set$new(1,2,3) # proper

ConditionalSet  Mathematical Set of Conditions

Description

A mathematical set defined by one or more logical conditions.

Details

Conditional sets are a useful tool for symbolically defining possibly infinite sets. They can be combined using standard 'and', &; and 'or', |, operators.

Super class

set6::Set -> ConditionalSet
Active bindings

- condition: Returns the condition defining the ConditionalSet.
- class: Returns argclass, see $new.
- elements: Returns NA.

Methods

**Public methods:**

- `ConditionalSet$new()`
- `ConditionalSet$contains()`
- `ConditionalSet$equals()`
- `ConditionalSet$strprint()`
- `ConditionalSet$summary()`
- `ConditionalSet$isSubset()`
- `ConditionalSet$clone()`

**Method** `new()`: Create a new `ConditionalSet` object.

**Usage:**

`ConditionalSet$new(condition = function(x) TRUE, argclass = NULL)`

**Arguments:**

- `condition`: function. Defines the set, see details.
- `argclass`: list. Optional list of sets that the function arguments live in, see details.

**Details:**
The condition should be given as a function that when evaluated returns either TRUE or FALSE. Further constraints can be given by providing the universe of the function arguments as Sets, if these are not given then Universal is assumed. See examples. Defaults construct the Universal set.

**Returns:** A new `ConditionalSet` object.

**Method** `contains()`: Tests to see if x is contained in the Set.

**Usage:**

`ConditionalSet$contains(x, all = FALSE, bound = NULL)`

**Arguments:**

- `x`: any. Object or vector of objects to test.
- `all`: logical. If FALSE tests each x separately. Otherwise returns TRUE only if all x pass test.
- `bound`: ignored, added for consistency.

**Details:** x can be of any type, including a Set itself. x should be a tuple if checking to see if it lies within a set of dimension greater than one. To test for multiple x at the same time, then provide these as a list.

If `all = TRUE` then returns TRUE if all x are contained in the Set, otherwise returns a vector of logicals.

An element is contained in a `ConditionalSet` if it returns TRUE as an argument in the defining function. For sets that are defined with a function that takes multiple arguments, a Tuple should be passed to x.
Returns: If all is TRUE then returns TRUE if all elements of x are contained in the Set, otherwise FALSE. If all is FALSE then returns a vector of logicals corresponding to each individual element of x.
The infix operator `%inset%` is available to test if x is an element in the Set, see examples.

Examples:

```r
# Set of positives
s = ConditionalSet$new(function(x) x > 0)
s$contains(list(1,-1))

# Set via equality
s = ConditionalSet$new(function(x, y) x + y == 2)
s$contains(list(Set$new(2, 0), Set$new(0, 2)))

# Tuples are recommended when using contains as they allow non-unique elements
s = ConditionalSet$new(function(x, y) x + y == 4)
\dontrun{
s$contains(Set$new(2, 2)) # Errors as Set$new(2,2) == Set$new(2)
}

# Set of Positive Naturals
s = ConditionalSet$new(function(x) TRUE, argclass = list(x = PosNaturals$new()))
s$contains(list(-2, 2))
```

Method equals(): Tests if two sets are equal.

Usage:
```r
ConditionalSet$equals(x, all = FALSE)
```

Arguments:
- `x` Set or vector of Sets.
- `all` logical. If FALSE tests each x separately. Otherwise returns TRUE only if all x pass test.

Details: Two sets are equal if they contain the same elements. Infix operators can be used for:

```
Equal       ==
Not equal   !=
```

Returns: If all is TRUE then returns TRUE if all x are equal to the Set, otherwise FALSE. If all is FALSE then returns a vector of logicals corresponding to each individual element of x.

Method strprint(): Creates a printable representation of the object.

Usage:
```r
ConditionalSet$strprint(n = NULL)
```

Arguments:
- `n` ignored, added for consistency.

Returns: A character string representing the object.

Method summary(): See strprint.
Usage:
ConditionalSet$summary(n = NULL)

Arguments:
n ignored, added for consistency.

Method isSubset(): Currently undefined for ConditionalSets.

Usage:
ConditionalSet$isSubset(x, proper = FALSE, all = FALSE)

Arguments:
x ignored, added for consistency.
proper ignored, added for consistency.
all ignored, added for consistency.

Method clone(): The objects of this class are cloneable with this method.

Usage:
ConditionalSet$clone(deep = FALSE)

Arguments:
deep Whether to make a deep clone.

See Also

Other sets: FuzzyMultiset, FuzzySet, FuzzyTuple, Interval, Multiset, Set, Tuple

Examples

# Set of Positive Naturals
s <- ConditionalSet$new(function(x) TRUE, argclass = list(x = PosNaturals$new()))

## ------------------------------------------------
## Method `ConditionalSet$contains`
## ------------------------------------------------

# Set of positives
s = ConditionalSet$new(function(x) x > 0)
s$contains(list(1,-1))

# Set via equality
s = ConditionalSet$new(function(x, y) x + y == 2)
s$contains(list(Set$new(2, 0), Set$new(0, 2)))

# Tuples are recommended when using contains as they allow non-unique elements
s = ConditionalSet$new(function(x, y) x + y == 4)
## Not run:
s$contains(Set$new(2, 2)) # Errors as Set$new(2,2) == Set$new(2)

## End(Not run)

# Set of Positive Naturals
s = ConditionalSet$new(function(x) TRUE, argclass = list(x = PosNaturals$new()))
s$contains(list(-2, 2))
contains  
contains Operator

Description
Operator for $contains methods. See Set$contains for full details. Operators can be used for:

<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
<th>Operator</th>
</tr>
</thead>
<tbody>
<tr>
<td>Contains</td>
<td>x contains y</td>
<td>$\text{y} \inset \text{x}$</td>
</tr>
</tbody>
</table>

Usage

\[
x \ %\text{inset}\% \ y
\]

Arguments

\[
x, \ y \quad \text{Set}
\]

equals  
equals Operator

Description
Operator for $equals methods. See Set$equals for full details. Operators can be used for:

<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
<th>Operator</th>
</tr>
</thead>
<tbody>
<tr>
<td>Equal</td>
<td>x equals y</td>
<td>$==$</td>
</tr>
<tr>
<td>Not Equal</td>
<td>x does not equal y</td>
<td>$!=$</td>
</tr>
</tbody>
</table>

Usage

\[
\#	ext{ S3 method for class 'Set'}
x \ == \ y
\]

\[
\#	ext{ S3 method for class 'Set'}
x \ != \ y
\]

Arguments

\[
x, \ y \quad \text{Set}
\]
ExponentSet

**Set of Exponentiations**

**Description**

ExponentSet class for symbolic exponentiation of mathematical sets.

**Details**

The purpose of this class is to provide a symbolic representation for the exponentiation of sets that cannot be represented in a simpler class. Whilst this is not an abstract class, it is not recommended to construct this class directly but via the set operation methods.

**Super classes**

`set6::Set` ➔ `set6::SetWrapper` ➔ `set6::ProductSet` ➔ ExponentSet

**Active bindings**

power Returns the power that the wrapped set is raised to.

**Methods**

**Public methods:**

- `ExponentSet$new()`
- `ExponentSet$strprint()`
- `ExponentSet$contains()`
- `ExponentSet$clone()`

**Method `new()`:** Create a new ExponentSet object. It is not recommended to construct this class directly.

*Usage:*

`ExponentSet$new(set, power)`

*Arguments:*

- `set` Set to wrap.
- `power` numeric. Power to raise Set to.

*Returns:* A new ExponentSet object.

**Method `strprint()`:** Creates a printable representation of the object.

*Usage:*

`ExponentSet$strprint(n = 2)`

*Arguments:*

- `n` numeric. Number of elements to display on either side of ellipsis when printing.

*Returns:* A character string representing the object.
Method `contains()`: Tests if elements \( x \) are contained in `self`.

**Usage:**
ExponentSet$contains(x, all = FALSE, bound = FALSE)

**Arguments:**
- \( x \) Set or vector of `Sets`.
- `all` logical. If `FALSE` tests each \( x \) separately. Otherwise returns `TRUE` only if all \( x \) pass test.
- `bound` logical

**Returns:** If `all == TRUE` then returns `TRUE` if all \( x \) are contained in `self`, otherwise `FALSE`. If `all == FALSE` returns a vector of logicals corresponding to the length of \( x \), representing if each is contained in `self`. If `bound == TRUE` then an element is contained in `self` if it is on or within the (possibly-open) bounds of `self`, otherwise `TRUE` only if the element is within `self` or the bounds are closed.

Method `clone()`: The objects of this class are cloneable with this method.

**Usage:**
ExponentSet$clone(deep = FALSE)

**Arguments:**
- `deep` Whether to make a deep clone.

See Also

Set operations: `setunion`, `setproduct`, `setpower`, `setcomplement`, `setsymdiff`, `powerset`, `setintersect`  
Other wrappers: `ComplementSet`, `PowersetSet`, `ProductSet`, `UnionSet`

---

ExtendedReals  

**Set of Extended Real Numbers**

**Description**

The mathematical set of extended real numbers, defined as the union of the set of reals with \( \pm \infty \).
i.e.

\[ R \cup -\infty, \infty \]

where \( R \) is the set of reals.

**Super classes**

`set6::Set` -> `set6::Interval` -> `set6::SpecialSet` -> `set6::Reals` -> `ExtendedReals`
Methods

Public methods:

• `ExtendedReals$new()`
• `ExtendedReals$clone()`

Method `new()`: Create a new `ExtendedReals` object.

Usage:

`ExtendedReals$new()`

Returns: A new `ExtendedReals` object.

Method `clone()`: The objects of this class are cloneable with this method.

Usage:

`ExtendedReals$clone(deep = FALSE)`

Arguments:

deep Whether to make a deep clone.

See Also

Other special sets: `Complex`, `Integers`, `Logicals`, `Naturals`, `NegIntegers`, `NegRationals`, `NegReals`, `PosIntegers`, `PosNaturals`, `PosRationals`, `PosReals`, `Rationals`, `Reals`, `Universal`
Methods

Public methods:

• `FuzzyMultiset$equals()`
• `FuzzyMultiset$isSubset()`
• `FuzzyMultiset$alphaCut()`
• `FuzzyMultiset$clone()`

Method `equals()`: Tests if two sets are equal.

Usage:
`FuzzyMultiset$equals(x, all = FALSE)`

Arguments:
- `x` Set or vector of Sets.
- `all` logical. If FALSE tests each `x` separately. Otherwise returns TRUE only if all `x` pass test.

Details: Two fuzzy sets are equal if they contain the same elements with the same memberships and in the same order. Infix operators can be used for:

```
Equal  ==
Not equal  !=
```

Returns: If `all` is TRUE then returns TRUE if all `x` are equal to the Set, otherwise FALSE. If `all` is FALSE then returns a vector of logicals corresponding to each individual element of `x`.

Method `isSubset()`: Test if one set is a (proper) subset of another

Usage:
`FuzzyMultiset$isSubset(x, proper = FALSE, all = FALSE)`

Arguments:
- `x` any. Object or vector of objects to test.
- `proper` logical. If TRUE tests for proper subsets.
- `all` logical. If FALSE tests each `x` separately. Otherwise returns TRUE only if all `x` pass test.

Details: If using the method directly, and not via one of the operators then the additional boolean argument `proper` can be used to specify testing of subsets or proper subsets. A Set is a proper subset of another if it is fully contained by the other Set (i.e. not equal to) whereas a Set is a (non-proper) subset if it is fully contained by, or equal to, the other Set. Infix operators can be used for:

```
Subset  <
Proper Subset  <=
Superset  >
Proper Superset  >=
```

Returns: If `all` is TRUE then returns TRUE if all `x` are subsets of the Set, otherwise FALSE. If `all` is FALSE then returns a vector of logicals corresponding to each individual element of `x`. 
Method alphaCut(): The alpha-cut of a fuzzy set is defined as the set

\[ A_\alpha = \{ x \in F | m \geq \alpha \} \]

where \( x \) is an element in the fuzzy set, \( F \), and \( m \) is the corresponding membership.

Usage:
FuzzyMultiset$alphaCut(alpha, strong = FALSE, create = FALSE)

Arguments:
alpha numeric in [0, 1] to determine which elements to return
strong logical, if FALSE (default) then includes elements greater than or equal to alpha, otherwise only strictly greater than
create logical, if FALSE (default) returns the elements in the alpha cut, otherwise returns a crisp set of the elements

Returns: Elements in FuzzyMultiset or a Set of the elements.

Method clone(): The objects of this class are cloneable with this method.

Usage:
FuzzyMultiset$clone(deep = FALSE)

Arguments:
deep Whether to make a deep clone.

See Also
Other sets: ConditionalSet, FuzzySet, FuzzyTuple, Interval, Multiset, Set, Tuple

Examples

# Different constructors
FuzzyMultiset$new(1, 0.5, 2, 1, 3, 0)
FuzzyMultiset$new(elements = 1:3, membership = c(0.5, 1, 0))

# Crisp sets are a special case FuzzyMultiset
# Note membership defaults to full membership
FuzzyMultiset$new(elements = 1:5) == Multiset$new(1:5)

f <- FuzzyMultiset$new(1, 0.2, 2, 1, 3, 0)
f$membership()
f$alphaCut(0.3)
f$core()
f$inclusion(0)
f$membership(0)
f$membership(1)

# Elements can be duplicated, and with different memberships,
# although this is not necessarily sensible.
FuzzyMultiset$new(1, 0.1, 1, 1)

# Like FuzzySets, ordering does not matter.
FuzzyMultiset$new(1, 0.1, 2, 0.2) == FuzzyMultiset$new(2, 0.2, 1, 0.1)
FuzzySet

Mathematical Fuzzy Set

Description

A general FuzzySet object for mathematical fuzzy sets, inheriting from Set.

Details

Fuzzy sets generalise standard mathematical sets to allow for fuzzy relationships. Whereas a standard, or crisp, set assumes that an element is either in a set or not, a fuzzy set allows an element to be in a set to a particular degree, known as the membership function, which quantifies the inclusion of an element by a number in \([0, 1]\). Thus a (crisp) set is a fuzzy set where all elements have a membership equal to 1. Similarly to Sets, elements must be unique and the ordering does not matter, to establish order and non-unique elements, FuzzyTuples can be used.

Super class

set6::Set -> FuzzySet

Methods

Public methods:

- FuzzySet$new()
- FuzzySet$strprint()
- FuzzySet$membership()
- FuzzySet$alphaCut()
- FuzzySet$support()
- FuzzySet$core()
- FuzzySet$inclusion()
- FuzzySet$equals()
- FuzzySet$isSubset()
- FuzzySet$clone()

Method new(): Create a new FuzzySet object.

Usage:

FuzzySet$new(
    ..., elements = NULL,
    membership = rep(1, length(elements)),
    class = NULL
)

Arguments:

... Alternating elements and membership, see details.
elements  Elements in the set, see details.
membership  Corresponding membership of the elements, see details.
class  Optional string naming a class that if supplied gives the set the typed property.

Details:  FuzzySets can be constructed in one of two ways, either by supplying the elements and their membership in alternate order, or by providing a list of elements to elements and a list of respective memberships to membership, see examples. If the class argument is non-NULL, then all elements will be coerced to the given class in construction, and if elements of a different class are added these will either be rejected or coerced.

Returns:  A new FuzzySet object.

Method strprint(): Creates a printable representation of the object.

Usage:
FuzzySet$strprint(n = 2)

Arguments:
  n numeric. Number of elements to display on either side of ellipsis when printing.

Returns:  A character string representing the object.

Method membership(): Returns the membership, i.e. value in [0, 1], of either the given element(s) or all elements in the fuzzy set.

Usage:
FuzzySet$membership(element = NULL)

Arguments:
  element  element or list of element in the set, if NULL returns membership of all elements

Details:  For FuzzySets this is straightforward and returns the membership of the given element(s), however in FuzzyTuples and FuzzyMultisets when an element may be duplicated, the function returns the membership of all instances of the element.

Returns:  Value, or list of values, in [0, 1].

Examples:
  f = FuzzySet$new(1, 0.1, 2, 0.5, 3, 1)
f$membership()
f$membership(2)
f$membership(list(1, 2))

Method alphaCut(): The alpha-cut of a fuzzy set is defined as the set

\[ A_\alpha = \{ x \in F | m(x) \geq \alpha \} \]

where \( x \) is an element in the fuzzy set, \( F \), and \( m \) is the corresponding membership.

Usage:
FuzzySet$alphaCut(alpha, strong = FALSE, create = FALSE)

Arguments:
  alpha  numeric in [0, 1] to determine which elements to return
  strong  logical, if FALSE (default) then includes elements greater than or equal to alpha, otherwise only strictly greater than
create logical, if FALSE (default) returns the elements in the alpha cut, otherwise returns a crisp set of the elements

*Returns*: Elements in FuzzySet or a Set of the elements.

*Examples*:

```r
f = FuzzySet$new(1, 0.1, 2, 0.5, 3, 1)
# Alpha-cut
f$alphaCut(0.5)

# Strong alpha-cut
f$alphaCut(0.5, strong = TRUE)

# Create a set from the alpha-cut
f$alphaCut(0.5, create = TRUE)
```

**Method** `support()`: The support of a fuzzy set is defined as the set of elements whose membership is greater than zero, or the strong alpha-cut with $\alpha = 0$,

$$A_\alpha = \{ x \in F | m > 0 \}$$

where $x$ is an element in the fuzzy set, $F$, and $m$ is the corresponding membership.

*Usage*:

```r
FuzzySet$support(create = FALSE)
```

*Arguments*:

- `create` logical, if FALSE (default) returns the support elements, otherwise returns a Set of the support elements

*Returns*: Support elements in fuzzy set or a Set of the support elements.

*Examples*:

```r
f = FuzzySet$new(0.1, 0, 1, 0.1, 2, 0.5, 3, 1)
f$support()
f$support(TRUE)
```

**Method** `core()`: The core of a fuzzy set is defined as the set of elements whose membership is equal to one, or the alpha-cut with $\alpha = 1$,

$$A_\alpha = \{ x \in F : m \geq 1 \}$$

where $x$ is an element in the fuzzy set, $F$, and $m$ is the corresponding membership.

*Usage*:

```r
FuzzySet$core(create = FALSE)
```

*Arguments*:

- `create` logical, if FALSE (default) returns the core elements, otherwise returns a Set of the core elements

*Returns*: Core elements in FuzzySet or a Set of the core elements.

*Examples*:  

f = FuzzySet$new(0.1, 0, 1, 0.1, 2, 0.5, 3, 1)
f$core()
f$core(TRUE)

**Method inclusion()**: An element in a fuzzy set, with corresponding membership \( m \), is:
- Included - If \( m = 1 \)
- Partially Included - If \( 0 < m < 1 \)
- Not Included - If \( m = 0 \)

**Usage**:
FuzzySet$inclusion(element)

**Arguments**:
- element  element or list of elements in fuzzy set for which to get the inclusion level

**Details**: For FuzzySets this is straightforward and returns the inclusion level of the given element(s), however in FuzzyTuples and FuzzyMultisets when an element may be duplicated, the function returns the inclusion level of all instances of the element.

**Returns**: One of: "Included", "Partially Included", "Not Included"

**Examples**:
f = FuzzySet$new(0.1, 0, 1, 0.1, 2, 0.5, 3, 1)
f$inclusion(0.1)
f$inclusion(1)
f$inclusion(3)

**Method equals()**: Tests if two sets are equal.

**Usage**:
FuzzySet$equals(x, all = FALSE)

**Arguments**:
- x  Set or vector of Sets.
- all  logical. If FALSE tests each x separately. Otherwise returns TRUE only if all x pass test.

**Details**: Two fuzzy sets are equal if they contain the same elements with the same memberships. Infix operators can be used for:

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Equal</td>
<td>==</td>
</tr>
<tr>
<td>Not equal</td>
<td>!=</td>
</tr>
</tbody>
</table>

**Returns**: If all is TRUE then returns TRUE if all x are equal to the Set, otherwise FALSE. If all is FALSE then returns a vector of logicals corresponding to each individual element of x.

**Method isSubset()**: Test if one set is a (proper) subset of another

**Usage**:
FuzzySet$isSubset(x, proper = FALSE, all = FALSE)

**Arguments**:
- x  any. Object or vector of objects to test.
proper logical. If TRUE tests for proper subsets. 

all logical. If FALSE tests each x separately. Otherwise returns TRUE only if all x pass test. 

Details: If using the method directly, and not via one of the operators then the additional boolean argument proper can be used to specify testing of subsets or proper subsets. A Set is a proper subset of another if it is fully contained by the other Set (i.e. not equal to) whereas a Set is a (non-proper) subset if it is fully contained by, or equal to, the other Set. 

Infix operators can be used for:

Subset < 
Proper Subset <= 
Superset > 
Proper Superset >= 

Returns: If all is TRUE then returns TRUE if all x are subsets of the Set, otherwise FALSE. If all is FALSE then returns a vector of logicals corresponding to each individual element of x. 

Method clone(): The objects of this class are cloneable with this method. 

Usage: 
FuzzySet$clone(deep = FALSE) 

Arguments: 
deep Whether to make a deep clone. 

See Also 
Other sets: ConditionalSet, FuzzyMultiset, FuzzyTuple, Interval, Multiset, Set, Tuple 

Examples 

# Different constructors 
FuzzySet$new(1, 0.5, 2, 1, 3, 0) 
FuzzySet$new(elements = 1:3, membership = c(0.5, 1, 0)) 

# Crisp sets are a special case FuzzySet 
# Note membership defaults to full membership 
FuzzySet$new(elements = 1:5) == Set$new(1:5) 

f <- FuzzySet$new(1, 0.2, 2, 1, 3, 0) 
f$membership() 
f$alphaCut(0.3) 
f$core() 
f$inclusion(0) 
f$membership(0) 
f$membership(1) 

## ------------------------------------------------ 
## Method \texttt{FuzzySet$membership} 
## ------------------------------------------------ 

f = FuzzySet$new(1, 0.1, 2, 0.5, 3, 1) 
f$membership()
```r
f$membership(2)
f$membership(list(1, 2))

# Method `FuzzySet$alphaCut`
# ------------------------------------------------
f = FuzzySet$new(1, 0.1, 2, 0.5, 3, 1)
# Alpha-cut
f$alphaCut(0.5)

# Strong alpha-cut
f$alphaCut(0.5, strong = TRUE)

# Create a set from the alpha-cut
f$alphaCut(0.5, create = TRUE)

# Method `FuzzySet$support`
# ------------------------------------------------
f = FuzzySet$new(0.1, 0, 1, 0.1, 2, 0.5, 3, 1)
f$support()
f$support(TRUE)

# Method `FuzzySet$core`
# ------------------------------------------------
f = FuzzySet$new(0.1, 0, 1, 0.1, 2, 0.5, 3, 1)
f$core()
f$core(TRUE)

# Method `FuzzySet$inclusion`
# ------------------------------------------------
f = FuzzySet$new(0.1, 0, 1, 0.1, 2, 0.5, 3, 1)
f$inclusion(0.1)
f$inclusion(1)
f$inclusion(3)
```

---

**FuzzyTuple**  

**Mathematical Fuzzy Tuple**

**Description**

A general FuzzyTuple object for mathematical fuzzy tuples, inheriting from FuzzySet.
Fuzzy tuples generalise standard mathematical tuples to allow for fuzzy relationships. Whereas a standard, or crisp, tuple assumes that an element is either in a tuple or not, a fuzzy tuple allows an element to be in a tuple to a particular degree, known as the membership function, which quantifies the inclusion of an element by a number in [0, 1]. Thus a (crisp) tuple is a fuzzy tuple where all elements have a membership equal to 1. Similarly to Tuples, elements do not need to be unique and the ordering does matter. FuzzySets are special cases where the ordering does not matter and elements must be unique.

Super classes

\[ \text{set6::Set} \rightarrow \text{set6::FuzzySet} \rightarrow \text{FuzzyTuple} \]

Methods

Public methods:

- FuzzyTuple\$equals()
- FuzzyTuple\$isSubset()
- FuzzyTuple\$alphaCut()
- FuzzyTuple\$clone()

Method equals(): Tests if two sets are equal.

Usage:
FuzzyTuple\$equals(x, all = FALSE)

Arguments:
x Set or vector of Sets.
all logical. If FALSE tests each x separately. Otherwise returns TRUE only if all x pass test.

Details: Two fuzzy sets are equal if they contain the same elements with the same memberships and in the same order. Infix operators can be used for:

\[ \text{Equal} \quad == \quad \text{Not equal} \quad != \]

Returns: If all is TRUE then returns TRUE if all x are equal to the Set, otherwise FALSE. If all is FALSE then returns a vector of logicals corresponding to each individual element of x.

Method isSubset(): Test if one set is a (proper) subset of another

Usage:
FuzzyTuple\$isSubset(x, proper = FALSE, all = FALSE)

Arguments:
x any. Object or vector of objects to test.
proper logical. If TRUE tests for proper subsets.
all logical. If FALSE tests each x separately. Otherwise returns TRUE only if all x pass test.
Details: If using the method directly, and not via one of the operators then the additional boolean argument proper can be used to specify testing of subsets or proper subsets. A Set is a proper subset of another if it is fully contained by the other Set (i.e. not equal to) whereas a Set is a (non-proper) subset if it is fully contained by, or equal to, the other Set. Infix operators can be used for:

- Subset <
- Proper Subset <=
- Superset >
- Proper Superset >=

Returns: If all is TRUE then returns TRUE if all x are subsets of the Set, otherwise FALSE. If all is FALSE then returns a vector of logicals corresponding to each individual element of x.

**Method alphaCut():** The alpha-cut of a fuzzy set is defined as the set

\[ A_\alpha = \{ x \in F | m \geq \alpha \} \]

where \( x \) is an element in the fuzzy set, \( F \), and \( m \) is the corresponding membership.

Usage:

FuzzyTuple$alphaCut(alpha, strong = FALSE, create = FALSE)

Arguments:

- alpha numeric in \([0, 1]\) to determine which elements to return
- strong logical, if FALSE (default) then includes elements greater than or equal to alpha, otherwise only strictly greater than
- create logical, if FALSE (default) returns the elements in the alpha cut, otherwise returns a crisp set of the elements

Returns: Elements in FuzzyTuple or a Set of the elements.

**Method clone():** The objects of this class are cloneable with this method.

Usage:

FuzzyTuple$clone(deep = FALSE)

Arguments:

- deep Whether to make a deep clone.

See Also

Other sets: ConditionalSet, FuzzyMultiset, FuzzySet, Interval, Multiset, Set, Tuple

Examples

# Different constructors
FuzzyTuple$new(1, 0.5, 2, 1, 3, 0)
FuzzyTuple$new(elements = 1:3, membership = c(0.5, 1, 0))

# Crisp sets are a special case FuzzyTuple
# Note membership defaults to full membership
FuzzyTuple$new(elements = 1:5) == Tuple$new(1:5)
f <- FuzzyTuple$new(1, 0.2, 2, 1, 3, 0)
f$membership()
f$alphaCut(0.3)
f$core()
f$inclusion(0)
f$membership(0)
f$membership(1)

# Elements can be duplicated, and with different memberships, although this is not necessarily sensible.
FuzzyTuple$new(1, 0.1, 1, 1)

# More important is ordering.
FuzzyTuple$new(1, 0.1, 2, 0.2) != FuzzyTuple$new(2, 0.2, 1, 0.1)
FuzzySet$new(1, 0.1, 2, 0.2) == FuzzySet$new(2, 0.2, 1, 0.1)

---

Integers

Set of Integers

Description

The mathematical set of integers, defined as the set of whole numbers. i.e.

..., −3, −2, −1, 0, 1, 2, 3, ...

Super classes

set6::Set -> set6::Interval -> set6::SpecialSet -> Integers

Methods

Public methods:

• Integers$new()
• Integers$clone()

Method new(): Create a new Integers object.

Usage:
Integers$new(lower = -Inf, upper = Inf, type = "()")

Arguments:
lower numeric. Where to start the set. Advised to ignore, used by child-classes.
upper numeric. Where to end the set. Advised to ignore, used by child-classes.
type character Set closure type. Advised to ignore, used by child-classes.

Returns: A new Integers object.

Method clone(): The objects of this class are cloneable with this method.

Usage:
Integers$clone(deep = FALSE)

Arguments:
deep Whether to make a deep clone.
See Also

Other special sets: Complex, ExtendedReals, Logicals, Naturals, NegIntegers, NegRationals, NegReals, PosIntegers, PosNaturals, PosRationals, PosReals, Rationals, Reals, Universal

---

Interval | Mathematical Finite or Infinite Interval

Description

A general Interval object for mathematical intervals, inheriting from Set. Intervals may be open, closed, or half-open; as well as bounded above, below, or not at all.

Details

The Interval class can be used for finite or infinite intervals, but often Sets will be preferred for integer intervals over a finite continuous range.

Super class

set6::Set -> Interval

Active bindings

length If the Interval is countably finite then returns the number of elements in the Interval, otherwise Inf. See the cardinality property for the type of infinity.

elements If the Interval is finite then returns all elements in the Interval, otherwise NA.

Methods

Public methods:

• Interval$new()
• Interval$strprint()
• Interval$equals()
• Interval$contains()
• Interval$isSubset()
• Interval$isSubinterval()
• Interval$clone()

Method new(): Create a new Interval object.

Usage:

```
Interval$new(
  lower = -Inf,
  upper = Inf,
  type = c("[", ",", "[", ",")
  class = "numeric",
  universe = ExtendedReals$new()
)
```
Interval

Arguments:
lower numeric. Lower limit of the interval.
upper numeric. Upper limit of the interval.
type character. One of: '()', '(]', '[)', '[]', which specifies if interval is open, left-open, right-open, or closed.
class character. One of: 'numeric', 'integer', which specifies if interval is over the Reals or Integers.
universe Set. Universe that the interval lives in, default Reals.

Details: Intervals are constructed by specifying the Interval limits, the boundary type, the class, and the possible universe. The universe differs from class as it is primarily used for the setcomplement method. Whereas class specifies if the interval takes integers or numerics, the universe specifies what range the interval could take.

Returns: A new Interval object.

Method strprint(): Creates a printable representation of the object.

Usage:
Interval$strprint(...)

Arguments:
... ignored, added for consistency.

Returns: A character string representing the object.

Method equals(): Tests if two sets are equal.

Usage:
Interval$equals(x, all = FALSE)

Arguments:
x Set or vector of Sets.
all logical. If FALSE tests each x separately. Otherwise returns TRUE only if all x pass test.

Details: Two Intervals are equal if they have the same: class, type, and bounds. Infix operators can be used for:

\[
\text{Equal} \quad == \\
\text{Not equal} \quad !=
\]

Returns: If all is TRUE then returns TRUE if all x are equal to the Set, otherwise FALSE. If all is FALSE then returns a vector of logicals corresponding to each individual element of x.

Examples:
Interval$new(1,5) == Interval$new(1,5)
Interval$new(1,5, class = "integer") != Interval$new(1,5, class="numeric")

Method contains(): Tests to see if x is contained in the Set.

Usage:
Interval$contains(x, all = FALSE, bound = FALSE)
Interval

Arguments:
x  any. Object or vector of objects to test.
all logical. If FALSE tests each x separately. Otherwise returns TRUE only if all x pass test.
bound logical.

Details:  x can be of any type, including a Set itself. x should be a tuple if checking to see if it lies within a set of dimension greater than one. To test for multiple x at the same time, then provide these as a list.
If all = TRUE then returns TRUE if all x are contained in the Set, otherwise returns a vector of logicals. For Intervals, bound is used to specify if elements lying on the (possibly open) boundary of the interval are considered contained (bound = TRUE) or not (bound = FALSE).

Returns:  If all is TRUE then returns TRUE if all elements of x are contained in the Set, otherwise FALSE. If all is FALSE then returns a vector of logicals corresponding to each individual element of x.
The infix operator %inset% is available to test if x is an element in the Set, see examples.

Examples:
s = Set$new(1:5)

# Simplest case
s$contains(4) 8 %inset% s

# Test if multiple elements lie in the set
s$contains(4:6, all = FALSE) s$contains(4:6, all = TRUE)

# Check if a tuple lies in a Set of higher dimension
s2 = s * s
s2$contains(Tuple$new(2,1)) c(Tuple$new(2,1), Tuple$new(1,7), 2) %inset% s2

Method isSubset(): Test if one set is a (proper) subset of another

Usage:
Interval$isSubset(x, proper = FALSE, all = FALSE)

Arguments:
x  any. Object or vector of objects to test.
proper logical. If TRUE tests for proper subsets.
all logical. If FALSE tests each x separately. Otherwise returns TRUE only if all x pass test.

Details:  If using the method directly, and not via one of the operators then the additional boolean argument proper can be used to specify testing of subsets or proper subsets. A Set is a proper subset of another if it is fully contained by the other Set (i.e. not equal to) whereas a Set is a (non-proper) subset if it is fully contained by, or equal to, the other Set.
When calling isSubset on objects inheriting from Interval, the method treats the interval as if it is a Set, i.e. ordering and class are ignored. Use isSubinterval to test if one interval is a subinterval of another.
Infix operators can be used for:
Returns: If all is TRUE then returns TRUE if all x are subsets of the Set, otherwise FALSE. If all is FALSE then returns a vector of logicals corresponding to each individual element of x.

Examples:
Interval$new(1,3) < Interval$new(1,5)
Set$new(1,3) < Interval$new(0,5)

Method isSubinterval(): Test if one interval is a (proper) subinterval of another

Usage:
Interval$isSubinterval(x, proper = FALSE, all = FALSE)

Arguments:
x Set or list
proper If TRUE then tests if x is a proper subinterval (i.e. subinterval and not equal to) of self, otherwise FALSE tests if x is a (non-proper) subinterval.
all If TRUE then returns TRUE if all x are subintervals, otherwise returns a vector of logicals.

Details: If x is a Set then will be coerced to an Interval if possible. $isSubinterval differs from $isSubset in that ordering and class are respected in $isSubinterval. See examples for a clearer illustration of the difference.

Returns: If all is TRUE then returns TRUE if all x are subsets of the Set, otherwise FALSE. If all is FALSE then returns a vector of logicals corresponding to each individual element of x.

Examples:
Interval$new(1,3)$isSubset(Set$new(1,2)) # TRUE
Interval$new(1,3)$isSubset(Set$new(2, 1)) # TRUE
Interval$new(1,3, class = "integer")$isSubinterval(Set$new(1,2)) # TRUE
Interval$new(1,3)$isSubinterval(Set$new(1, 2)) # FALSE
Interval$new(1,3)$isSubinterval(Set$new(2, 1)) # FALSE

Reals$new()$isSubset(Integers$new()) # TRUE
Reals$new()$isSubinterval(Integers$new()) # FALSE

Method clone(): The objects of this class are cloneable with this method.

Usage:
Interval$clone(deep = FALSE)

Arguments:
deep Whether to make a deep clone.

See Also

Other sets: ConditionalSet, FuzzyMultiset, FuzzySet, FuzzyTuple, Multiset, Set, Tuple
Examples

# Set of Reals
Interval$new()

# Set of Integers
Interval$new(class = "integer")

# Half-open interval
i <- Interval$new(1, 10, "["")
i$contains(c(1, 10))
i$contains(c(1, 10), bound = TRUE)

# Equivalent Set and Interval
Set$new(1:5) == Interval$new(1, 5, class = "integer")

# SpecialSets can provide more efficient implementation
Interval$new() == ExtendedReals$new()
Interval$new(class = "integer", type = "()") == Integers$new()

## Method \(\text{Interval$equals}\)

Interval$new(1,5) == Interval$new(1,5)
Interval$new(1,5, class = "integer") != Interval$new(1,5,class="numeric")

## Method \(\text{Interval$contains}\)

s = Set$new(1:5)

# Simplest case
s$contains(4)
8 %inset% s

# Test if multiple elements lie in the set
s$contains(4:6, all = FALSE)
s$contains(4:6, all = TRUE)

# Check if a tuple lies in a Set of higher dimension
s2 = s * s
s2$contains(Tuple$new(2,1))
c(Tuple$new(2,1), Tuple$new(1,7), 2) %inset% s2

## Method \(\text{Interval$isSubset}\)

Interval$new(1,3) < Interval$new(1,5)
Set$new(1,3) < Interval$new(0,5)
## isSubset

### Description

Operator for $isSubset$ methods. See Set$sisSubset$ for full details. Operators can be used for:

<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
<th>Operator</th>
</tr>
</thead>
<tbody>
<tr>
<td>Subset</td>
<td>x is a subset of y</td>
<td>x &lt;= y</td>
</tr>
<tr>
<td>Proper Subset</td>
<td>x is a proper subset of y</td>
<td>x &lt; y</td>
</tr>
<tr>
<td>Superset</td>
<td>x is a superset of y</td>
<td>x &gt;= y</td>
</tr>
<tr>
<td>Proper Superset</td>
<td>x is a proper superset of y</td>
<td>x &gt; y</td>
</tr>
</tbody>
</table>

### Usage

```r
## S3 method for class 'Set'
x < y

## S3 method for class 'Set'
x <= y

## S3 method for class 'Set'
x > y

## S3 method for class 'Set'
x >= y
```

### Arguments

- `x, y` : Set
### listSpecialSets

**Lists Implemented R6 Special Sets**

**Description**
Lists special sets that can be used in Set.

**Usage**

```r
listSpecialSets(simplify = FALSE)
```

**Arguments**

- `simplify` logical. If `FALSE` (default) returns data.frame of set name and symbol, otherwise set names as characters.

**Value**

Either a list of characters (if `simplify` is `TRUE`) or a data.frame of SpecialSets and their traits.

**Examples**

```r
listSpecialSets()
listSpecialSets(TRUE)
```

---

### Logicals

**Set of Logicals**

**Description**
The `Logicals` is defined as the `Set` containing the elements `TRUE` and `FALSE`.

**Super class**

```
set6::Set -> Logicals
```

**Methods**

**Public methods:**
- `Logicals$new()`
- `Logicals$clone()`

**Method `new()`**: Create a new `Logicals` object.

**Usage**

```r
Logicals$new()
```
**LogicalSet**

*Details:* The Logical set is the set containing `TRUE` and `FALSE`.

*Returns:* A new `Logicals` object.

**Method** `clone()`: The objects of this class are cloneable with this method.

*Usage:*

```r
Logicals$clone(deep = FALSE)
```

*Arguments:*

- `deep` Whether to make a deep clone.

**See Also**

Other special sets: `Complex`, `ExtendedReals`, `Integers`, `Naturals`, `NegIntegers`, `NegRationals`, `NegReals`, `PosIntegers`, `PosNaturals`, `PosRationals`, `PosReals`, `Rationals`, `Reals`, `Universal`

**Examples**

```r
l <- LogicalSet$new()
p <- l$contains(list(TRUE, 1, FALSE))
```

---

**LogicalSet**  
*Set of Logicals*

**Description**

The LogicalSet is defined as the Set containing the elements `TRUE` and `FALSE`.

**Super class**

`set6::Set` -> LogicalSet

**Methods**

**Public methods:**

- `LogicalSet$new()`  
- `LogicalSet$clone()`

**Method** `new()`: Create a new LogicalSet object.

*Usage:*

```r
LogicalSet$new()
```

*Details:* The Logical set is the set containing `TRUE` and `FALSE`.

*Returns:* A new LogicalSet object.

**Method** `clone()`: The objects of this class are cloneable with this method.

*Usage:*

```r
LogicalSet$clone(deep = FALSE)
```

*Arguments:*

- `deep` Whether to make a deep clone.
Examples

```r
l <- LogicalSet$new()
print(l)
l$contains(list(TRUE, 1, FALSE))
```

Description

A general Multiset object for mathematical Multisets, inheriting from Set.

Details

Multisets are generalisations of sets that allow an element to be repeated. They can be thought of as Tuples without ordering.

Super class

`set6::Set` -> `Multiset`

Methods

Public methods:

- `Multiset$equals()`
- `Multiset$isSubset()`
- `Multiset$clone()`

Method `equals()`: Tests if two sets are equal.

Usage:

```r
Multiset$equals(x, all = FALSE)
```

Arguments:

- `x` Set or vector of Sets.
- `all` logical. If FALSE tests each x separately. Otherwise returns TRUE only if all x pass test.

Details: An object is equal to a Multiset if it contains all the same elements, and in the same order. Infix operators can be used for:

```
Equal     ==
Not equal !=
```

Returns: If all is TRUE then returns TRUE if all x are equal to the Set, otherwise FALSE. If all is FALSE then returns a vector of logicals corresponding to each individual element of x.

Examples:

```r
Multiset$new(1,2) == Multiset$new(1,2)
```
Multiset$new(1,2) != Multiset$new(1,2)
Multiset$new(1,1) != Set$new(1,1)

Method isSubset(): Test if one set is a (proper) subset of another

Usage:
Multiset$isSubset(x, proper = FALSE, all = FALSE)

Arguments:
x any. Object or vector of objects to test.
proper logical. If TRUE tests for proper subsets.
all logical. If FALSE tests each x separately. Otherwise returns TRUE only if all x pass test.

Details: If using the method directly, and not via one of the operators then the additional boolean argument proper can be used to specify testing of subsets or proper subsets. A Set is a proper subset of another if it is fully contained by the other Set (i.e. not equal to) whereas a Set is a (non-proper) subset if it is fully contained by, or equal to, the other Set.

When calling SisSubset on objects inheriting from Interval, the method treats the interval as if it is a Set, i.e. ordering and class are ignored. Use $isSubinterval to test if one interval is a subinterval of another.

Infix operators can be used for:

<table>
<thead>
<tr>
<th>Subset</th>
<th>&lt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>Proper Subset</td>
<td>&lt;=</td>
</tr>
<tr>
<td>Superset</td>
<td></td>
</tr>
<tr>
<td>Proper Superset</td>
<td>&gt;=</td>
</tr>
</tbody>
</table>

An object is a (proper) subset of a Multiset if it contains all (some) of the same elements, and in the same order.

Returns: If all is TRUE then returns TRUE if all x are subsets of the Set, otherwise FALSE. If all is FALSE then returns a vector of logicals corresponding to each individual element of x.

Examples:
Multiset$new(1,2,3) < Multiset$new(1,2,3,4)
Multiset$new(1,3,2) < Multiset$new(1,2,3,4)
Multiset$new(1,3,2,4) <= Multiset$new(1,2,3,4)

Method clone(): The objects of this class are cloneable with this method.

Usage:
Multiset$clone(deep = FALSE)

Arguments:

| deep | Whether to make a deep clone. |

See Also
Other sets: ConditionalSet, FuzzyMultiset, FuzzySet, FuzzyTuple, Interval, Set, Tuple
Examples

# Multiset of integers
Multiset$new(1:5)

# Multiset of multiple types
Multiset$new("a", 5, Set$new(1), Multiset$new(2))

# Each Multiset has properties and traits
t <- Multiset$new(1, 2, 3)
t$traits
t$properties

# Elements can be duplicated
Multiset$new(2, 2) != Multiset$new(2)

# Ordering does not matter
Multiset$new(1, 2) == Multiset$new(2, 1)

## ------------------------------------------------
## Method ‘Multiset$equals’
## ------------------------------------------------
Multiset$new(1,2) == Multiset$new(1,2)
Multiset$new(1,2) != Multiset$new(1,2)
Multiset$new(1,1) != Set$new(1,1)

## ------------------------------------------------
## Method ‘Multiset$isSubset’
## ------------------------------------------------
Multiset$new(1,2,3) < Multiset$new(1,2,3,4)
Multiset$new(1,3,2) < Multiset$new(1,2,3,4)
Multiset$new(1,3,2,4) <= Multiset$new(1,2,3,4)

Naturals

Set of Natural Numbers

Description

The mathematical set of natural numbers, defined as the counting numbers. i.e.

0, 1, 2, ...

Super classes

set6::Set -> set6::Interval -> set6::SpecialSet -> Naturals
NegIntegers

Methods

Public methods:

• Naturals$new()
• Naturals$clone()

Method new(): Create a new Naturals object.

Usage:
Naturals$new(lower = 0)

Arguments:
lower numeric. Where to start the set. Advised to ignore, used by child-classes.

Returns: A new Naturals object.

Method clone(): The objects of this class are cloneable with this method.

Usage:
Naturals$clone(deep = FALSE)

Arguments:
deep Whether to make a deep clone.

See Also

Other special sets: Complex, ExtendedReals, Integers, Logicals, NegIntegers, NegRationals,
NegReals, PosIntegers, PosNaturals, PosRationals, PosReals, Rationals, Reals, Universal
NegIntegers$new(zero = FALSE)

Arguments:
zero  logical. If TRUE, zero is included in the set.

Returns:  A new NegIntegers object.

Method clone():  The objects of this class are cloneable with this method.

Usage:
NegIntegers$clone(deep = FALSE)

Arguments:
deep  Whether to make a deep clone.

See Also
Other special sets:  Complex, ExtendedReals, Integers, Logicals, Naturals, NegRationals, NegReals, PosIntegers, PosNaturals, PosRationals, PosReals, Rationals, Reals, Universal

NegRationals  Set of Negative Rational Numbers

Description
The mathematical set of negative rational numbers, defined as the set of numbers that can be written as a fraction of two integers and are non-positive. i.e.

\[ \frac{p}{q} : p, q \in \mathbb{Z}, p/q \leq 0, q \neq 0 \]

where \( \mathbb{Z} \) is the set of integers.

Details
The $contains method does not work for the set of Rationals as it is notoriously difficult/impossible to find an algorithm for determining if any given number is rational or not. Furthermore, computers must truncate all irrational numbers to rational numbers.

Super classes

\[ \text{set6::Set} \rightarrow \text{set6::Interval} \rightarrow \text{set6::SpecialSet} \rightarrow \text{set6::Rationals} \rightarrow \text{NegRationals} \]

Methods

Public methods:
- NegRationals$new()
- NegRationals$clone()

Method new():  Create a new NegRationals object.
NegReals

Usage:
NegRationals$new(zero = FALSE)

Arguments:
zero logical. If TRUE, zero is included in the set.

Returns: A new NegRationals object.

Method clone(): The objects of this class are cloneable with this method.

Usage:
NegRationals$clone(deep = FALSE)

Arguments:
deep Whether to make a deep clone.

See Also

Other special sets: Complex, ExtendedReals, Integers, Logicals, Naturals, NegIntegers, NegReals, PosIntegers, PosNaturals, PosRationals, PosReals, Rationals, Reals, Universal

NegReals

Set of Negative Real Numbers

Description

The mathematical set of negative real numbers, defined as the union of the set of negative rationals and negative irrationals. i.e.

\[ I^- \cup Q^- \]

where \( I^- \) is the set of negative irrationals and \( Q^- \) is the set of negative rationals.

Super classes

set6::Set -> set6::Interval -> set6::SpecialSet -> set6::Reals -> NegReals

Methods

Public methods:
• NegReals$new()
• NegReals$clone()

Method new(): Create a new NegReals object.

Usage:
NegReals$new(zero = FALSE)

Arguments:
zero logical. If TRUE, zero is included in the set.

Returns: A new NegReals object.
**Method** `clone()`: The objects of this class are cloneable with this method.

*Usage:*
```r
NegReals$clone(deep = FALSE)
```

*Arguments:*
- `deep`  Whether to make a deep clone.

**See Also**

Other special sets: `Complex`, `ExtendedReals`, `Integers`, `Logicals`, `Naturals`, `NegIntegers`, `NegRationals`, `PosIntegers`, `PosNaturals`, `PosRationals`, `PosReals`, `Rationals`, `Reals`, `Universal

---

**PosIntegers**

*Set of Positive Integers*

**Description**

The mathematical set of positive integers, defined as the set of positive whole numbers. i.e.

\[ 0, 1, 2, 3, ... \]

**Super classes**

`set6::Set` -> `set6::Interval` -> `set6::SpecialSet` -> `set6::Integers` -> `PosIntegers`

**Methods**

**Public methods:**
- `PosIntegers$new()`
- `PosIntegers$clone()`

**Method** `new()`: Create a new `PosIntegers` object.

*Usage:*
```r
PosIntegers$new(zero = FALSE)
```

*Arguments:*
- `zero`  logical. If TRUE, zero is included in the set.

*Returns:* A new `PosIntegers` object.

**Method** `clone()`: The objects of this class are cloneable with this method.

*Usage:*
```r
PosIntegers$clone(deep = FALSE)
```

*Arguments:*
- `deep`  Whether to make a deep clone.

**See Also**

Other special sets: `Complex`, `ExtendedReals`, `Integers`, `Logicals`, `Naturals`, `NegIntegers`, `NegRationals`, `NegReals`, `PosNaturals`, `PosRationals`, `PosReals`, `Rationals`, `Reals`, `Universal`
**PosNaturals**  

*Set of Positive Natural Numbers*

---

**Description**

The mathematical set of positive natural numbers, defined as the positive counting numbers. i.e.

$1, 2, 3, \ldots$

**Super classes**

`set6::Set -> set6::Interval -> set6::SpecialSet -> set6::Naturals -> PosNaturals`

**Methods**

**Public methods:**

- `PosNaturals$new()`
- `PosNaturals$clone()`

**Method** `new()`: Create a new `PosNaturals` object.

*Usage:*

`PosNaturals$new()`

*Returns:* A new `PosNaturals` object.

**Method** `clone()`: The objects of this class are cloneable with this method.

*Usage:*

`PosNaturals$clone(deep = FALSE)`

*Arguments:*

- `deep` Whether to make a deep clone.

**See Also**

Other special sets: Complex, ExtendedReals, Integers, Logicals, Naturals, NegIntegers, NegRationals, NegReals, PosIntegers, PosRationals, PosReals, Rationals, Reals, Universal
PosRationals

**Set of Positive Rational Numbers**

**Description**

The mathematical set of positive rational numbers, defined as the set of numbers that can be written as a fraction of two integers and are non-negative. i.e.

\[
\frac{p}{q} : p, q \in \mathbb{Z}, \frac{p}{q} \geq 0, q \neq 0
\]

where \( \mathbb{Z} \) is the set of integers.

**Details**

The `$contains` method does not work for the set of Rationals as it is notoriously difficult/impossible to find an algorithm for determining if any given number is rational or not. Furthermore, computers must truncate all irrational numbers to rational numbers.

**Super classes**

set6::Set -> set6::Interval -> set6::SpecialSet -> set6::Rationals -> PosRationals

**Methods**

**Public methods:**

- PosRationals$new()
- PosRationals$clone()

**Method** `new()`: Create a new PosRationals object.

*Usage:*

PosRationals$new(zero = FALSE)

*Arguments:*

zero logical. If TRUE, zero is included in the set.

*Returns:* A new PosRationals object.

**Method** `clone()`: The objects of this class are cloneable with this method.

*Usage:*

PosRationals$clone(deep = FALSE)

*Arguments:*

deep Whether to make a deep clone.

**See Also**

Other special sets: Complex, ExtendedReals, Integers, Logicals, Naturals, NegIntegers, NegRationals, NegReals, PosIntegers, PosNaturals, PosReals, Rationals, Reals, Universal
**PosReals**

Set of Positive Real Numbers

**Description**

The mathematical set of positive real numbers, defined as the union of the set of positive rationals and positive irrationals. i.e.

\[ I^+ \cup Q^+ \]

where \( I^+ \) is the set of positive irrationals and \( Q^+ \) is the set of positive rationals.

**Super classes**

set6::Set -> set6::Interval -> set6::SpecialSet -> set6::Reals -> PosReals

**Methods**

**Public methods:**

- PosReals$new()
- PosReals$clone()

**Method new():** Create a new PosReals object.

*Usage:*

```r
PosReals$new(zero = FALSE)
```

*Arguments:*

- `zero` logical. If TRUE, zero is included in the set.

*Returns:* A new PosReals object.

**Method clone():** The objects of this class are cloneable with this method.

*Usage:*

```r
PosReals$clone(deep = FALSE)
```

*Arguments:*

- `deep` Whether to make a deep clone.

**See Also**

Other special sets: Complex, ExtendedReals, Integers, Logicals, Naturals, NegIntegers, NegRationals, NegReals, PosIntegers, PosNaturals, PosRationals, Rationals, Reals, Universal
powerset

Calculate a Set's Powerset

Description

Calculates and returns the powerset of a Set.

Usage

powerset(x, simplify = FALSE)

Arguments

x
Set

simplify
logical, if TRUE then tries to simplify the result to a Set otherwise creates an object of class PowersetSet.

Details

A powerset of a set, S, is defined as the set of all subsets of S, including S itself and the empty set.

Value

Set

See Also

Other operators: setcomplement(), setintersect(), setpower(), setproduct(), setsymdiff(), setunion()

Examples

# simplify = FALSE is default
powerset(Set$new(1, 2))
powerset(Set$new(1, 2), simplify = TRUE)

# powerset of intervals
powerset(Interval$new())

# powerset of powersets
powerset(powerset(Reals$new()))
powerset(powerset(Reals$new()))$properties$cardinality
PowersetSet

Set of Powersets

Description

PowersetSet class for symbolic powerset of mathematical sets.

Details

The purpose of this class is to provide a symbolic representation for the powerset of sets that cannot be represented in a simpler class. Whilst this is not an abstract class, it is not recommended to construct this class directly but via the set operation methods.

Super classes

```
set6::Set -> set6::SetWrapper -> set6::ProductSet -> PowersetSet
```

Methods

Public methods:

• PowersetSet$new()
• PowersetSet$strprint()
• PowersetSet$contains()
• PowersetSet$isSubset()
• PowersetSet$clone()

Method new(): Create a new PowersetSet object. It is not recommended to construct this class directly.

Usage:

```
PowersetSet$new(set)
```

Arguments:

set Set to wrap.

Returns: A new PowersetSet object.

Method strprint(): Creates a printable representation of the object.

Usage:

```
PowersetSet$strprint(n = 2)
```

Arguments:

n numeric. Number of elements to display on either side of ellipsis when printing.

Returns: A character string representing the object.

Method contains(): Tests if elements x are contained in self.

Usage:
PowersetSet$contains(x, all = FALSE, bound = NULL)

Arguments:
  x  Set or vector of Sets.
  all logical. If FALSE tests each x separately. Otherwise returns TRUE only if all x pass test.
  bound logical

Returns: If all == TRUE then returns TRUE if all x are contained in self, otherwise FALSE. If all == FALSE returns a vector of logicals corresponding to the length of x, representing if each is contained in self. If bound == TRUE then an element is contained in self if it is on or within the (possibly-open) bounds of self, otherwise TRUE only if the element is within self or the bounds are closed.

Method isSubset(): Tests if x is a (proper) subset of self.

Usage:
PowersetSet$isSubset(x, proper = FALSE, all = FALSE)

Arguments:
  x  Set or vector of Sets.
  proper logical. If TRUE tests for proper subsets.
  all logical. If FALSE tests each x separately. Otherwise returns TRUE only if all x pass test.

Returns: If all == TRUE then returns TRUE if all x are (proper) subsets of self, otherwise FALSE. If all == FALSE returns a vector of logicals corresponding to the length of x, representing if each is a (proper) subset of self.

Method clone(): The objects of this class are cloneable with this method.

Usage:
PowersetSet$clone(deep = FALSE)

Arguments:
  deep Whether to make a deep clone.

See Also
Set operations: setunion, setproduct, setpower, setcomplement, setsymdiff, powerset, setintersect
Other wrappers: ComplementSet, ExponentSet, ProductSet, UnionSet
Description

ProductSet class for symbolic product of mathematical sets.

Details

The purpose of this class is to provide a symbolic representation for the product of sets that cannot be represented in a simpler class. Whilst this is not an abstract class, it is not recommended to construct this class directly but via the set operation methods.

Super classes

set6::Set -> set6::SetWrapper -> ProductSet

Active bindings

length Returns the number of elements in the object.

Methods

Public methods:

• ProductSet$new()
• ProductSet$strprint()
• ProductSet$contains()
• ProductSet$clone()

Method new(): Create a new ProductSet object. It is not recommended to construct this class directly.

Usage:
ProductSet$new(
    setlist,
    lower = NULL,
    upper = NULL,
    type = NULL,
    cardinality = NULL
)

Arguments:
setlist list of Sets to wrap.
lower lower bound of new object.
upper upper bound of new object.
type closure type of new object.
cardinality: Either an integer, "Aleph0", or a beth number. If NULL then calculated automatically (recommended).

Returns: A new ProductSet object.

**Method strprint():** Creates a printable representation of the object.

*Usage:*

ProductSet$strprint(n = 2)

*Arguments:*

n numeric. Number of elements to display on either side of ellipsis when printing.

*Returns: A character string representing the object.*

**Method contains():** Tests if elements x are contained in self.

*Usage:*

ProductSet$contains(x, all = FALSE, bound = FALSE)

*Arguments:*

x Set or vector of Sets.

all logical. If FALSE tests each x separately. Otherwise returns TRUE only if all x pass test.

bound logical

*Returns:* If all == TRUE then returns TRUE if all x are contained in self, otherwise FALSE. If all == FALSE returns a vector of logicals corresponding to the length of x, representing if each is contained in self. If bound == TRUE then an element is contained in self if it is on or within the (possibly-open) bounds of self, otherwise TRUE only if the element is within self or the bounds are closed.

**Method clone():** The objects of this class are cloneable with this method.

*Usage:*

ProductSet$clone(deep = FALSE)

*Arguments:*

deep Whether to make a deep clone.

See Also

Set operations: setunion, setproduct, setpower, setcomplement, setsymdiff, powerset, setintersect

Other wrappers: ComplementSet, ExponentSet, PowersetSet, UnionSet
Properties

Set Properties Class

Description

Used to store the properties of a Set. Though this is not an abstract class, it should never be constructed outside of the Set constructor.

Active bindings

closure  Returns the closure of the Set. One of "open", "half-open", or "closed."
countability  Returns the countability of the Set. One of "countably finite", "countably infinite", or "uncountable".
cardinality  Returns the cardinality of the Set. Either an integer if the Set is countably finite, Aleph0 if countably infinite, or a Beth number.
empty  Returns if the Set is empty or not. TRUE if the Set cardinality is 0, FALSE otherwise.
singleton  Returns if the Set is a singleton or not. TRUE if the Set cardinality is 1, FALSE otherwise.

Methods

Public methods:

- Properties$new()
- Properties$print()
- Properties$strprint()
- Properties$clone()

Method new(): Creates a new Properties object.

Usage:
Properties$new(closure = character(0), cardinality = NULL)

Arguments:
closure  One of "open", "half-open", or "closed."
cardinality  If non-NULL then either an integer, "Aleph0", or a Beth number.

Returns:  A new Properties object.

Method print(): Prints the Properties list.

Usage:
Properties$print()

Returns:  Prints Properties list to console.

Method strprint(): Creates a printable representation of the Properties.

Usage:
Properties$strprint()
**Rationals**

**Returns:** A list of properties.

**Method** `clone()`: The objects of this class are cloneable with this method.

**Usage:**

Properties$clone(deep = FALSE)

**Arguments:**

deep Whether to make a deep clone.

---

**Description**

The mathematical set of rational numbers, defined as the set of numbers that can be written as a fraction of two integers. i.e.

\[
\frac{p}{q} : p, q \in \mathbb{Z}, q \neq 0
\]

where \( \mathbb{Z} \) is the set of integers.

**Details**

The $contains method does not work for the set of Rationals as it is notoriously difficult/impossible to find an algorithm for determining if any given number is rational or not. Furthermore, computers must truncate all irrational numbers to rational numbers.

**Super classes**

`set6::Set` -&gt; `set6::Interval` -&gt; `set6::SpecialSet` -&gt; Rationals

**Methods**

**Public methods:**

- Rationals$new()
- Rationals$contains()
- Rationals$isSubset()
- Rationals$equals()
- Rationals$clone()

**Method** `new()`: Create a new Rationals object.

**Usage:**

Rationals$new(lower = -Inf, upper = Inf, type = "()")

**Arguments:**

lower numeric. Where to start the set. Advised to ignore, used by child-classes.

upper numeric. Where to end the set. Advised to ignore, used by child-classes.
type  character  Set closure type. Advised to ignore, used by child-classes.

_Returns_: A new Rationals object.

**Method** `contains()`: Method not possible for Rationals.

_Usage:_
Rationals$contains(...)

_Arguments:_
... Ignored

**Method** `isSubset()`: Method not possible for Rationals.

_Usage:_
Rationals$isSubset(...)

_Arguments:_
... Ignored

**Method** `equals()`: Method not possible for Rationals.

_Usage:_
Rationals$equals(...)

_Arguments:_
... Ignored

**Method** `clone()`: The objects of this class are cloneable with this method.

_Usage:_
Rationals$clone(deep = FALSE)

_Arguments:_
depth Whether to make a deep clone.

**See Also**

Other special sets: Complex, ExtendedReals, Integers, Logicals, Naturals, NegIntegers, NegRationals, NegReals, PosIntegers, PosNaturals, PosRationals, PosReals, Reals, Universal

---

| Reals | Set of Real Numbers |

**Description**

The mathematical set of real numbers, defined as the union of the set of rationals and irrationals. i.e.

\[ I \cup Q \]

where \( I \) is the set of irrationals and \( Q \) is the set of rationals.
Super classes

\[
\text{set6::Set} \rightarrow \text{set6::Interval} \rightarrow \text{set6::SpecialSet} \rightarrow \text{Reals}
\]

Methods

Public methods:

- \text{Reals$new()}
- \text{Reals$clone()}

Method \text{new()}: Create a new Reals object.

\textbf{Usage:}

\text{Reals$new(lower = -Inf, upper = Inf, type = "()")}

\textbf{Arguments:}

lower numeric. Where to start the set. Advised to ignore, used by child-classes.
upper numeric. Where to end the set. Advised to ignore, used by child-classes.
type character Set closure type. Advised to ignore, used by child-classes.

\textbf{Returns:} A new Reals object.

Method \text{clone()}: The objects of this class are cloneable with this method.

\textbf{Usage:}

\text{Reals$clone(deep = FALSE)}

\textbf{Arguments:}

deep Whether to make a deep clone.

See Also

Other special sets: Complex, ExtendedReals, Integers, Logicals, Naturals, NegIntegers, NegRationals, NegReals, PosIntegers, PosNaturals, PosRationals, PosReals, Rationals, Universal

---

Set  \hspace{1cm} \textit{Mathematical Set}

Description

A general Set object for mathematical sets. This also serves as the parent class to intervals, tuples, and fuzzy variants.

Details

Mathematical sets can loosely be thought of as a collection of objects of any kind. The Set class is used for sets of finite elements, for infinite sets use Interval. These can be expanded for fuzzy logic by using FuzzySets. Elements in a set cannot be duplicated and ordering of elements does not matter, Tuples can be used if duplicates or ordering are required.
Active bindings

properties Returns an object of class Properties, which lists the properties of the Set. Set properties include:
• empty - is the Set empty or does it contain elements?
• singleton - is the Set a singleton? i.e. Does it contain only one element?
• cardinality - number of elements in the Set
• countability - One of: countably finite, countably infinite, uncountable
• closure - One of: closed, open, half-open

traits List the traits of the Set. Set traits include:
• crisp - is the Set crisp or fuzzy?

max If the Set consists of numerics only then returns the maximum element in the Set. For open or half-open sets, then the maximum is defined by
\[ upper - 1e - 15 \]

min If the Set consists of numerics only then returns the minimum element in the Set. For open or half-open sets, then the minimum is defined by
\[ lower + 1e - 15 \]

upper If the Set consists of numerics only then returns the upper bound of the Set.

lower If the Set consists of numerics only then returns the lower bound of the Set.

class If all elements in the Set are the same class then returns that class, otherwise "ANY".

elements If the Set is finite then returns all elements in the Set as a list, otherwise "NA".

universe Returns the universe of the Set, i.e. the set of values that can be added to the Set.

range If the Set consists of numerics only then returns the range of the Set defined by
\[ upper - lower \]

length If the Set is finite then returns the number of elements in the Set, otherwise Inf. See the cardinality property for the type of infinity.

Methods

Public methods:
• Set$new()
• Set$print()
• Set$strprint()
• Set$summary()
• Set[contains()]
• Set$equals()
• Set$isSubset()
• Set$add()
- **Set$remove()**
- **Set$multiplicity()**
- **Set$clone()**

**Method new():** Create a new Set object.

*Usage:*

Set$new(..., universe = Universal$new(), elements = NULL, class = NULL)

*Arguments:*

... ANY Elements can be of any class except list, as long as there is a unique as.character coercion method available.

universe: Set. Universe that the Set lives in, i.e. elements that could be added to the Set. Default is Universal.
elements: list. Alternative constructor that may be more efficient if passing objects of multiple classes.
class: character. Optional string naming a class that if supplied gives the set the typed property. All elements will be coerced to this class and therefore there must be a coercion method to this class available.

*Returns:* A new Set object.

**Method print():** Prints a symbolic representation of the Set.

*Usage:*

Set$print(n = 2)

*Arguments:*

n: numeric. Number of elements to display on either side of ellipsis when printing.

*Details:* The function useUnicode() can be used to determine if unicode should be used when printing the Set. Internally print first calls strprint to create a printable representation of the Set.

**Method strprint():** Creates a printable representation of the object.

*Usage:*

Set$strprint(n = 2)

*Arguments:*

n: numeric. Number of elements to display on either side of ellipsis when printing.

*Returns:* A character string representing the object.

**Method summary():** Summarises the Set.

*Usage:*

Set$summary(n = 2)

*Arguments:*

n: numeric. Number of elements to display on either side of ellipsis when printing.

*Details:* The function useUnicode() can be used to determine if unicode should be used when printing the Set. Summarised details include the Set class, properties, and traits.
Method `contains()`: Tests to see if \( x \) is contained in the `Set`.

Usage:

\[
\text{Set} \text{$\text{contains}(x, \; \text{all} = \text{FALSE}, \; \text{bound} = \text{NULL})}\]

Arguments:

- \( x \) any object or vector of objects to test.
- \( \text{all} \) logical. If \( \text{FALSE} \) tests each \( x \) separately. Otherwise returns \( \text{TRUE} \) only if all \( x \) pass test.
- \( \text{bound} \) ignored, added for consistency.

Details: \( x \) can be of any type, including a `Set` itself. \( x \) should be a tuple if checking to see if it lies within a set of dimension greater than one. To test for multiple \( x \) at the same time, then provide these as a list.

If \( \text{all} = \text{TRUE} \) then returns \( \text{TRUE} \) if all \( x \) are contained in the `Set`, otherwise returns a vector of logicals.

Returns: If \( \text{all} \) is \( \text{TRUE} \) then returns \( \text{TRUE} \) if all elements of \( x \) are contained in the `Set`, otherwise \( \text{FALSE} \). If \( \text{all} \) is \( \text{FALSE} \) then returns a vector of logicals corresponding to each individual element of \( x \).

The infix operator `%inset%` is available to test if \( x \) is an element in the `Set`, see examples.

Examples:

\[
\text{s} = \text{Set}$\text{new}(\text{elements} = 1:5)\\
\]

# Simplest case
\[
\text{s}$\text{contains}(4)\\
8 \%\text{inset}\% \text{s}\\
\]

# Test if multiple elements lie in the set
\[
\text{s}$\text{contains}(4:6, \; \text{all} = \text{FALSE})\\
\text{s}$\text{contains}(4:6, \; \text{all} = \text{TRUE})\\
\]

# Check if a tuple lies in a Set of higher dimension
\[
\text{s2} = \text{s} \* \text{s}\\
\text{s2}$\text{contains}($\text{Tuple}$\text{new}(2,1))\\
1 (\text{Tuple}$\text{new}(2,1), \; \text{Tuple}$\text{new}(1,7), \; 2) \%\text{inset}\% \text{s2}\\
\]

Method `equals()`: Tests if two sets are equal.

Usage:

\[
\text{Set}$\text{equals}(x, \; \text{all} = \text{FALSE})\\
\]

Arguments:

- \( x \) `Set` or vector of `Sets`.
- \( \text{all} \) logical. If \( \text{FALSE} \) tests each \( x \) separately. Otherwise returns \( \text{TRUE} \) only if all \( x \) pass test.

Details: Two sets are equal if they contain the same elements. Infix operators can be used for:

\[
\text{Equal} \; = \\
\text{Not equal} \; !=
\]
Returns: If all is TRUE then returns TRUE if all x are equal to the Set, otherwise FALSE. If all is FALSE then returns a vector of logicals corresponding to each individual element of x.

Examples:

# Equals
Set$new(1,2)$equals(Set$new(5,6))
Set$new(1,2)$equals(Interval$new(1,2))
Set$new(1,2) == Interval$new(1,2, class = "integer")

# Not equal
!Set$new(1,2)$equals(Set$new(1,2))
Set$new(1,2) ! = Set$new(1,5)

Method isSubset(): Test if one set is a (proper) subset of another

Usage:
Set$isSubset(x, proper = FALSE, all = FALSE)

Arguments:

x any. Object or vector of objects to test.
proper logical. If TRUE tests for proper subsets.
all logical. If FALSE tests each x separately. Otherwise returns TRUE only if all x pass test.

Details: If using the method directly, and not via one of the operators then the additional boolean argument proper can be used to specify testing of subsets or proper subsets. A Set is a proper subset of another if it is fully contained by the other Set (i.e. not equal to) whereas a Set is a (non-proper) subset if it is fully contained by, or equal to, the other Set.

Infix operators can be used for:

Subset            <
Proper Subset     <=
Superset           >
Proper Superset   >=

Returns: If all is TRUE then returns TRUE if all x are subsets of the Set, otherwise FALSE. If all is FALSE then returns a vector of logicals corresponding to each individual element of x.

Examples:

Set$new(1,2,3)$isSubset(Set$new(1,2), proper = TRUE)
Set$new(1,2) < Set$new(1,2,3) # proper subset

c(Set$new(1,2,3), Set$new(1)) < Set$new(1,2,3) # not proper
Set$new(1,2,3) <= Set$new(1,2,3) # proper

Method add(): Add elements to a set.

Usage:
Set$add(...)

Arguments:

... elements to add
$add$ is a wrapper around the $setunion$ method with $setunion(self, Set$new(...))$. Note a key difference is that any elements passed to ... are first converted to a Set, this important difference is illustrated in the examples by adding an $Interval$ to a Set.
Additionally, $add$ first coerces ... to $class$ if self is a typed-set (i.e. $class$ != "ANY"), and $add$ checks if elements in ... live in the universe of self.

$Returns$: An object inheriting from Set.

$Examples$:
Set$new(1,2)$add(3)$print()
Set$new(1,2,universe = Interval$new(1,3))$add(3)$print()
\dontrun{
# errors as 4 is not in [1,3]
Set$new(1,2,universe = Interval$new(1,3))$add(4)$print()
}
# coerced to complex
Set$new(0+1i, 2i, class = "complex")$add(4)$print()

# setunion vs. add
Set$new(1,2)$add(Interval$new(5,6))$print()
Set$new(1,2) + Interval$new(5,6)

$Method remove()$: Remove elements from a set.

$Usage$:
Set$remove(...)$

$Arguments$:
... elements to remove

$Details$: $remove$ is a wrapper around the $setcomplement$ method with $setcomplement(self, Set$new(...))$. Note a key difference is that any elements passed to ... are first converted to a Set, this important difference is illustrated in the examples by removing an $Interval$ from a Set.

$Returns$: If the complement cannot be simplified to a Set then a ComplementSet is returned otherwise an object inheriting from Set is returned.

$Examples$:
Set$new(1,2,3)$remove(1,2)$print()
Set$new(1,Set$new(1),2)$remove(Set$new(1))$print()
Interval$new(1,5)$remove(5)$print()
Interval$new(1,5)$remove(4)$print()

# setcomplement vs. remove
Set$new(1,2,3)$remove(Interval$new(5,7))$print()
Set$new(1,2,3) - Interval$new(5,7)

$Method multiplicity()$: Returns the number of times an element appears in a set.

$Usage$:
Set$multiplicity(element = NULL)$

$Arguments$:
element element or list of elements in the set, if NULL returns multiplicity of all elements
Returns: Value, or list of values, in R+.

Examples:

```
Set$new(1, 1, 2)$multiplicity()
Set$new(1, 1, 2)$multiplicity(1)
Set$new(1, 1, 2)$multiplicity(list(1, 2))
Tuple$new(1, 1, 2)$multiplicity()
Tuple$new(1, 1, 2)$multiplicity(1)
Tuple$new(1, 1, 2)$multiplicity(2)
```

Method `clone()`: The objects of this class are cloneable with this method.

Usage:

```
Set$clone(deep = FALSE)
```

Arguments:

- **deep** Whether to make a deep clone.

See Also

Other sets: `ConditionalSet`, `FuzzyMultiset`, `FuzzySet`, `FuzzyTuple`, `Interval`, `Multiset`, `Tuple`

Examples

```
# Set of integers
Set$new(1:5)

# Set of multiple types
Set$new("a", 5, Set$new(1))

# Each Set has properties and traits
s <- Set$new(1, 2, 3)
s$traits
s$properties

# Elements cannot be duplicated
Set$new(2, 2) == Set$new(2)

# Ordering does not matter
Set$new(1, 2) == Set$new(2, 1)
```

```
## ------------------------------------------------
## Method `\$contains`
## ------------------------------------------------
```

`s = Set$new(elements = 1:5)`

# Simplest case
`s$contains(4)`
8 %inset% s

# Test if multiple elements lie in the set
s$contains(4:6, all = FALSE)`
s$contains(4:6, all = TRUE)

# Check if a tuple lies in a Set of higher dimension
s2 = s * s
s2$contains(Tuple$new(2,1))
c(Tuple$new(2,1), Tuple$new(1,7), 2) %inset% s2

## Method `Set$equals`

# Equals
Set$new(1,2)$equals(Set$new(5,6))
Set$new(1,2)$equals(Interval$new(1,2))
Set$new(1,2) == Interval$new(1,2, class = "integer")

# Not equal
!Set$new(1,2)$equals(Set$new(1,2))
Set$new(1,2) != Set$new(1,5)

## Method `Set$isSubset`

Set$new(1,2,3)$isSubset(Set$new(1,2), proper = TRUE)
Set$new(1,2) < Set$new(1,2,3) # proper subset
c(Set$new(1,2,3), Set$new(1)) < Set$new(1,2,3) # not proper
Set$new(1,2,3) <= Set$new(1,2,3) # proper

## Method `Set$add`

Set$new(1,2)$add(3)$print()
Set$new(1,2,universe = Interval$new(1,3))$add(3)$print()

## End(Not run)
# coerced to complex
Set$new(0+1i, 2i, class = "complex")$add(4)$print()

# setunion vs. add
Set$new(1,2)$add(Interval$new(5,6))$print()
Set$new(1,2) + Interval$new(5,6)

## Method `Set$remove`

Set$new(1,2,3)$remove(1,2)$print()
setcomplement

Description

Returns the set difference of two objects inheriting from class Set. If y is missing then the complement of x from its universe is returned.

set6News

Show set6 NEWS.md File

Description

Displays the contents of the NEWS.md file for viewing set6 release information.

Usage

set6News()

Value

NEWS.md in viewer.

Examples

set6News()
Usage

setcomplement(x, y, simplify = TRUE)

## S3 method for class 'Set'
setcomplement(x, y, simplify = TRUE)

## S3 method for class 'Interval'
setcomplement(x, y, simplify = TRUE)

## S3 method for class 'FuzzySet'
setcomplement(x, y, simplify = TRUE)

## S3 method for class 'ConditionalSet'
setcomplement(x, y, simplify = TRUE)

## S3 method for class 'Reals'
setcomplement(x, y, simplify = TRUE)

## S3 method for class 'Rationals'
setcomplement(x, y, simplify = TRUE)

## S3 method for class 'Integers'
setcomplement(x, y, simplify = TRUE)

## S3 method for class 'ComplementSet'
setcomplement(x, y, simplify = TRUE)

## S3 method for class 'Set'
setcomplement(x, y, simplify = TRUE)

Arguments

x, y: Set
simplify: logical, if TRUE (default) returns the result in its simplest form, usually a Set or UnionSet, otherwise a ComplementSet.

Details

The difference of two sets, \( X, Y \), is defined as the set of elements that exist in set \( X \) and not \( Y \),

\[
X - Y = \{ z : z \in X \text{ and } \neg (z \in Y) \}
\]

The set difference of two ConditionalSets is defined by combining their defining functions by a negated 'and', \(!\&\), operator. See examples.

The complement of fuzzy and crisp sets first coerces fuzzy sets to crisp sets by finding their support.

Value

An object inheriting from Set containing the set difference of elements in \( x \) and \( y \).
setintersect

Intersection of Two Sets

Description

Returns the intersection of two objects inheriting from class Set.

See Also

Other operators: powerset(), setintersect(), setpower(), setproduct(), setsymdiff(), setunion()
setintersect

Usage

setintersect(x, y)

## S3 method for class 'Interval'
setintersect(x, y)

## S3 method for class 'ConditionalSet'
setintersect(x, y)

## S3 method for class 'UnionSet'
setintersect(x, y)

## S3 method for class 'ComplementSet'
setintersect(x, y)

## S3 method for class 'ProductSet'
setintersect(x, y)

## S3 method for class 'Set'
x & y

Arguments

x, y
Set

Details

The intersection of two sets, $X, Y$, is defined as the set of elements that exist in both sets,

$$X \cap Y = \{z : z \in X \text{ and } z \in Y\}$$

In the case where no elements are common to either set, then the empty set is returned.

The intersection of two ConditionalSets is defined by combining their defining functions by an 'and', & operator. See examples.

The intersection of fuzzy and crisp sets first coerce fuzzy sets to crisp sets by finding their support.

Value

A Set consisting of elements in both x and y.

See Also

Other operators: powerset(), setcomplement(), setpower(), setproduct(), setsymdiff(), setunion()

Examples

# intersection of two sets
setpower = Power of a Set

Description

A convenience wrapper for the n-ary cartesian product of a Set by itself, possibly multiple times.

Usage

setpower(x, power, simplify = FALSE, nest = FALSE)

## S3 method for class 'Set'

x ^ power

Arguments

x Set
setpower

power
power to raise set to, if "n" then a variable dimension set is created, see examples.
simplify
logical, if TRUE returns the result in its simplest (unwrapped) form, usually a Set, otherwise an ExponentSet.
nest
logical, if FALSE (default) returns the n-ary cartesian product, otherwise returns the cartesian product applied n times. Sets. See details and examples.

Details
See the details of setproduct for a longer discussion on the use of the nest argument, in particular with regards to n-ary cartesian products vs. 'standard' cartesian products.

Value
An R6 object of class Set or ExponentSet inheriting from ProductSet.

See Also
Other operators: powerset(), setcomplement(), setintersect(), setproduct(), setsymdiff(), setunion()

Examples
# Power of a Set
setpower(Set$new(1, 2), 3, simplify = FALSE)
setpower(Set$new(1, 2), 3, simplify = TRUE)
Set$new(1, 2)^3

# Power of an interval
Interval$new(2, 5)^5
Reals$new()^3

# Use tuples for contains
(PosNaturals$new()^3)$contains(Tuple$new(1, 2, 3))

# Power of ConditionalSet is meaningless
ConditionalSet$new(function(x) TRUE)^2

# Power of FuzzySet
FuzzySet$new(1, 0.1, 2, 0.5)^2

# Variable length
x <- Interval$new(0, 1)^"n"
x$contains(Tuple$new(0))
x$contains(Tuple$new(0, 1))
x$contains(Tuple$new(0, 1, 0, 1, 1))
x$contains(list(Tuple$new(0, 2), Tuple$new(1, 1)))
setproduct  

Cartesian Product of Sets

Description

Returns the cartesian product of objects inheriting from class Set.

Usage

setproduct(..., simplify = FALSE, nest = FALSE)

## S3 method for class 'Set'

x * y

Arguments

...  
Sets

simplify  
logical, if TRUE returns the result in its simplest (unwrapped) form, usually a Set otherwise a ProductSet.

nest  
logical, if FALSE (default) then will treat any ProductSets passed to ... as unwrapped Sets. See details and examples.

x, y  
Set

Details

The cartesian product of multiple sets, the 'n-ary Cartesian product', is often implemented in programming languages as being identical to the cartesian product of two sets applied recursively. However, for sets \(X, Y, Z\),

\[XYZ \neq (XY)Z\]

This is accommodated with the nest argument. If nest == TRUE then \(X \times Y \times Z \equiv (XY)Z\), i.e. the cartesian product for two sets is applied recursively. If nest == FALSE then \(X \times Y \times Z \equiv (XYZ)\) and the n-ary cartesian product is computed. As it appears the latter (n-ary product) is more common, nest = FALSE is the default. The N-ary cartesian product of \(N\) sets, \(X_1, ..., X_N\), is defined as

\[X_1...X_N = (x_1, ..., x_N) : x_1 \in X_1 \cap \ldots \cap x_N \in X_N\]

where \((x_1, ..., x_N)\) is a tuple.

The product of fuzzy and crisp sets first coerce fuzzy sets to crisp sets by finding their support.

Value

Either an object of class ProductSet or an unwrapped object inheriting from Set.

See Also

Other operators: powerset(), setcomplement(), setintersect(), setpower(), setsymdiff(), setunion()
Examples

# difference between nesting
Set$new(1, 2) * Set$new(2, 3) * Set$new(4, 5)
setproduct(Set$new(1, 2) * Set$new(2, 3), Set$new(4, 5), nest = FALSE)  # same as above
setproduct(Set$new(1, 2) * Set$new(2, 3), Set$new(4, 5), nest = TRUE)
unnest_set <- setproduct(Set$new(1, 2) * Set$new(2, 3), Set$new(4, 5), nest = FALSE)
nest_set <- setproduct(Set$new(1, 2) * Set$new(2, 3), Set$new(4, 5), nest = TRUE)
# note the difference when using contains
unnest_set$contains(Tuple$new(1, 3, 5))
nest_set$contains(Tuple$new(Tuple$new(1, 3), 5))

# product of two sets
Set$new(-2:4) * Set$new(2:5)
setproduct(Set$new(1, 4, "a"), Set$new("a", 6))
setproduct(Set$new(1, 4, "a"), Set$new("a", 6), simplify = TRUE)

# product of two intervals
Interval$new(1, 10) * Interval$new(5, 15)
Interval$new(1, 2, type = "()") * Interval$new(2, 3, type = "["
Interval$new(1, 5, class = "integer") *
  Interval$new(2, 7, class = "integer")

# product of mixed set types
Set$new(1:10) * Interval$new(5, 15)
Set$new(5, 7) * Tuple$new(6, 8, 7)
FuzzySet$new(1, 0.1) * Set$new(2)

# product of FuzzySet
FuzzySet$new(1, 0.1, 2, 0.5) * Set$new(2:5)

# product of conditional sets
ConditionalSet$new(function(x, y) x >= y) *
  ConditionalSet$new(function(x, y) x == y)

# product of special sets
PosReals$new() * NegReals$new()
Arguments

- **x, y**: Set
- **simplify**: logical, if TRUE (default) returns the result in its simplest form, usually a Set or UnionSet, otherwise a ComplementSet.

Details

The symmetric difference, aka disjunctive union, of two sets, \( X, Y \), is defined as the set of elements that exist in set \( X \) or in \( Y \) but not both,

\[
\{ z : (z \in X \cup z \in Y) \cap \lnot (z \in X \cap z \in Y) \}
\]

The symmetric difference can also be expressed as the union of two sets minus the intersection. Therefore `setsymdiff` is written as a thin wrapper over these operations, so for two sets, \( A, B \):

\[
A \%-% B = (A | B) - (A & B).
\]

The symmetric difference of fuzzy and crisp sets first coerces fuzzy sets to crisp sets by finding their support.

Value

An object inheriting from `Set` containing the symmetric difference of elements in both \( x \) and \( y \).

See Also

Other operators: `powerset()`, `setcomplement()`, `setintersect()`, `setpower()`, `setproduct()`, `setunion()`

Examples

```r
# symmetrical difference compared to union and intersection
Set$new(1, 2, 3) %-% Set$new(3, 4)
(Set$new(1, 2, 3) | Set$new(3, 4)) - (Set$new(1, 2, 3) & Set$new(3, 4))

# ConditionalSets demonstrate the internal logic
ConditionalSet$new(function(x) x > 0) %-%
ConditionalSet$new(function(y) y == 0)
```

---

**setunion**

*Union of Sets*

Description

Returns the union of objects inheriting from class `Set`. 
setunion

Usage

setunion(..., simplify = TRUE)

## S3 method for class 'Set'
 x + y

## S3 method for class 'Set'
 x | y

Arguments

...  Sets
simplify  logical, if TRUE (default) returns the result in its simplest (unwrapped) form, usually a Set, otherwise a UnionSet.
x, y  Set

Details

The union of \(N\) sets, \(X_1, \ldots, X_N\), is defined as the set of elements that exist in one or more sets,

\[
U = \{x : \text{either } x \in X_1 \text{ or } x \in X_2 \text{ or } \ldots \text{ or } x \in X_N \}\]

The union of multiple ConditionalSets is given by combining their defining functions by an 'or', |, operator. See examples.
The union of fuzzy and crisp sets first coerces fuzzy sets to crisp sets by finding their support.

Value

An object inheriting from Set containing the union of supplied sets.

See Also

Other operators: powerset(), setcomplement(), setintersect(), setpower(), setproduct(), setsymdiff()

Examples

# union of Sets
Set$new(-2:4) + Set$new(2:5)
setunion(Set$new(1, 4, "a"), Set$new("a", 6))
Set$new(1, 2) + Set$new("a", 1i) + Set$new(9)

# union of intervals
Interval$new(1, 10) + Interval$new(5, 15) + Interval$new(20, 30)
Interval$new(1, 2, type = "()") + Interval$new(2, 3, type = "[")
Interval$new(1, 5, class = "integer") +
    Interval$new(2, 7, class = "integer")
# union of mixed types

Set$new(1:10) + Interval$new(5, 15)
Set$new(1:10) + Interval$new(5, 15, class = "integer")
Set$new(5, 7) | Tuple$new(6, 8, 7)

# union of FuzzySet
FuzzySet$new(1, 0.1, 2, 0.5) + Set$new(2:5)

# union of conditional sets
ConditionalSet$new(function(x, y) x >= y) +
  ConditionalSet$new(function(x, y) x == y) +
  ConditionalSet$new(function(x) x == 2)

# union of special sets
PosReals$new() + NegReals$new()
Set$new(-Inf, Inf) + Reals$new()

---

**SetWrapper**

*Abstract SetWrapper Class*

**Description**

This class should not be constructed directly. Parent class to SetWrappers.

**Details**

Wrappers in set6 are utilised to facilitate lazy evaluation and symbolic representation. Each operation has an associated wrapper that will be returned if `simplify = FALSE` or if the result would be too complex to return as a simple Set. Wrappers have an identical interface to Set. Their primary advantage lies in a neat representation of any set composition (the result of an operation) and the ability to query the set contents without ever directly evaluating the set elements.

**Super class**

`set6::Set` -> SetWrapper

**Active bindings**

`wrappedSets` Returns the list of Sets that are wrapped in the given wrapper.

**Methods**

*Public methods:*
  - `SetWrapper$new()`
  - `SetWrapper$equals()`
SetWrapper

- `SetWrapper$isNewSubSet()`
- `SetWrapper$clone()`

**Method** `new()`: Create a new `SetWrapper` object. It is not recommended to construct this class directly.

**Usage:**
```
SetWrapper$new(
    setlist, 
    lower = NULL, 
    upper = NULL, 
    type = NULL, 
    class = NULL, 
    cardinality
)
```

**Arguments:**
- `setlist`: List of Sets to wrap.
- `lower`: Set. Lower bound of wrapper.
- `upper`: Set. Upper bound of wrapper.
- `type`: character. Closure type of wrapper.
- `class`: character. Ignored.
- `cardinality`: character or integer. Cardinality of wrapper.

**Returns:** A new `SetWrapper` object.

**Method** `equals()`: Tests if `x` is equal to `self`.

**Usage:**
```
SetWrapper$equals(x, all = FALSE)
```

**Arguments:**
- `x`: Set or vector of Sets.
- `all`: logical. If FALSE tests each `x` separately. Otherwise returns TRUE only if all `x` pass test.

**Returns:**
- If `all` == TRUE then returns TRUE if all `x` are equal to `self`, otherwise FALSE. If `all` == FALSE returns a vector of logicals corresponding to the length of `x`, representing if each is equal to `self`.

**Method** `isSubset()`: Tests if `x` is a (proper) subset of `self`.

**Usage:**
```
SetWrapper$isNewSubSet(x, proper = FALSE, all = FALSE)
```

**Arguments:**
- `x`: Set or vector of Sets.
- `all`: logical. If FALSE tests each `x` separately. Otherwise returns TRUE only if all `x` pass test.
- `proper`: logical. If TRUE tests for proper subsets.
all logical. If FALSE tests each x separately. Otherwise returns TRUE only if all x pass test.

Returns: If all == TRUE then returns TRUE if all x are (proper) subsets of self, otherwise FALSE. If all == FALSE returns a vector of logicals corresponding to the length of x, representing if each is a (proper) subset of self.

Method clone(): The objects of this class are cloneable with this method.

Usage:
SetWrapper$clone(deep = FALSE)

Arguments:
deep Whether to make a deep clone.

Description
The ‘special sets’ are the group of sets that are commonly used in mathematics and are thus given their own names.

Details
This is an abstract class and should not be constructed directly. Use listSpecialSets to see the list of implemented special sets.

Super classes
set6::Set -> set6::Interval -> SpecialSet

Methods

Public methods:
• SpecialSet$new()
• SpecialSet$strprint()
• SpecialSet$clone()

Method new(): SpecialSet is an abstract class, the constructor cannot be used directly.

Usage:
SpecialSet$new(lower = -Inf, upper = Inf, type = "\(\)", class = "numeric")

Arguments:
lower defines the lower bound of the interval.
upper defines the upper bound of the interval.
type defines the interval closure type.
class defines the interval class.
**Method** `strprint()`: Creates a printable representation of the object.

*Usage:*

`SpecialSet$strprint(n = NULL)`

*Arguments:*

- `n`: ignored, added for consistency.

*Returns:* A character string representing the object.

**Method** `clone()`: The objects of this class are cloneable with this method.

*Usage:*

`SpecialSet$clone(deep = FALSE)`

*Arguments:*

- `deep`: Whether to make a deep clone.

---

**Description**

Validation checks to test if a given object is closed.

**Usage**

```r
testClosed(object, errormsg = "This is not a closed set")
checkClosed(object, errormsg = "This is not a closed set")
assertClosed(object, errormsg = "This is not a closed set")
```

**Arguments**

- `object`: object to test
- `errormsg`: error message to overwrite default if check fails

**Value**

If check passes then assert returns object invisibly and test/check return TRUE. If check fails, assert stops code with error, check returns an error message as string, and test returns FALSE.

**Examples**

```r
testClosed(Interval$new(1, 10, type = "["))
testClosed(Interval$new(1, 10, type = "]"))
```
testClosedAbove

**Description**
Validation checks to test if a given object is closed above.

**Usage**
```r
testClosedAbove(object, errormsg = "This is not a set closed above")
checkClosedAbove(object, errormsg = "This is not a set closed above")
assertClosedAbove(object, errormsg = "This is not a set closed above")
```

**Arguments**
- **object**
  - object to test
- **errormsg**
  - error message to overwrite default if check fails

**Value**
If check passes then assert returns object invisibly and test/check return TRUE. If check fails, assert stops code with error, check returns an error message as string, and test returns FALSE.

**Examples**
```r
testClosedAbove(Interval$new(1, 10, type = "["))
testClosedAbove(Interval$new(1, 10, type = "["))
```

testClosedBelow

**Description**
Validation checks to test if a given object is closed below.

**Usage**
```r
testClosedBelow(object, errormsg = "This is not a set closed below")
checkClosedBelow(object, errormsg = "This is not a set closed below")
assertClosedBelow(object, errormsg = "This is not a set closed below")
```
Arguments

- `object`: object to test
- `errormsg`: error message to overwrite default if check fails

Value

If check passes then `assert` returns object invisibly and `test/check` return `TRUE`. If check fails, `assert` stops code with error, `check` returns an error message as string, and `test` returns `FALSE`.

Examples

```r
testClosedBelow(Interval$new(1, 10, type = "[]"))
```

```r
testClosedBelow(Interval$new(1, 10, type = "(]"))
```

Description

Validation checks to test if a given object is an R6 ConditionalSet.

Usage

```r
testConditionalSet(
  object, 
  errormsg = "This is not an R6 ConditionalSet object"
)
```

```r
checkConditionalSet(
  object, 
  errormsg = "This is not an R6 ConditionalSet object"
)
```

```r
assertConditionalSet(
  object, 
  errormsg = "This is not an R6 ConditionalSet object"
)
```

Arguments

- `object`: object to test
- `errormsg`: error message to overwrite default if check fails

Value

If check passes then `assert` returns object invisibly and `test/check` return `TRUE`. If check fails, `assert` stops code with error, `check` returns an error message as string, and `test` returns `FALSE`.
Examples

testConditionalSet(Set$new(2, 3))
testConditionalSet(list(Set$new(2), Set$new(3)))
testConditionalSet(Tuple$new(2, 3))
testConditionalSet(Interval$new())
testConditionalSet(FuzzySet$new(2, 0.1))
testConditionalSet(FuzzyTuple$new(2, 0.1))
testConditionalSet(ConditionalSet$new(function(x) x == 0))

testContains

Description

Validation checks to test if given elements are contained in a set.

Usage

testContains(
    object,
    elements,
    errmsg = "elements are not contained in the set"
)

cHECKContains(
    object,
    elements,
    errmsg = "elements are not contained in the set"
)

assertContains(
    object,
    elements,
    errmsg = "elements are not contained in the set"
)

Arguments

object object to test

elements elements to check

errmsg error message to overwrite default if check fails

Value

If check passes then assert returns object invisibly and test/check return TRUE. If check fails, assert stops code with error, check returns an error message as string, and test returns FALSE.
Examples

testContains(Set$new(1,2,3), c(1,2))
testContains(Set$new(1,2,3), c(3,4))

testCountablyFinite assert/check/test/CountablyFinite

Description

Validation checks to test if a given object is countably finite.

Usage

testCountablyFinite(object, errmsg = "This is not a countably finite set")
checkCountablyFinite(object, errmsg = "This is not a countably finite set")
assertCountablyFinite(object, errmsg = "This is not a countably finite set")

Arguments

object object to test
errmsg error message to overwrite default if check fails

Value

If check passes then assert returns object invisibly and test/check return TRUE. If check fails, assert stops code with error, check returns an error message as string, and test returns FALSE.

Examples

testCountablyFinite(Set$new(1,2,3))
testCountablyFinite(Interval$new(1,10))

testCrisp assert/check/test/Crisp

Description

Validation checks to test if a given object is crisp.
Usage

```r
testCrisp(object, errmsg = "This is not crisp.")
checkCrisp(object, errmsg = "This is not crisp.")
assertCrisp(object, errmsg = "This is not crisp.")
```

Arguments

- **object**: object to test
- **errmsg**: error message to overwrite default if check fails

Value

If check passes then `assert` returns `object` invisibly and `test/check` return `TRUE`. If check fails, `assert` stops code with error, `check` returns an error message as string, and `test` returns `FALSE`.

Examples

```r
testCrisp(Set$new(1))
testCrisp(FuzzySet$new(1, 0.5))
```

---

### testEmpty

**Description**

Validation checks to test if a given object is empty.

**Usage**

```r
testEmpty(object, errmsg = "This is not an empty set")
checkEmpty(object, errmsg = "This is not an empty set")
assertEmpty(object, errmsg = "This is not an empty set")
```

Arguments

- **object**: object to test
- **errmsg**: error message to overwrite default if check fails

Value

If check passes then `assert` returns `object` invisibly and `test/check` return `TRUE`. If check fails, `assert` stops code with error, `check` returns an error message as string, and `test` returns `FALSE`. 
testFinite

Examples
  testEmpty(Set$new())
  testEmpty(Set$new(1))

Description
  Validation checks to test if a given object is finite.

Usage
  testFinite(object, errmsg = "This is not finite")
  checkFinite(object, errmsg = "This is not finite")
  assertFinite(object, errmsg = "This is not finite")

Arguments
  object  object to test
  errmsg  error message to overwrite default if check fails

Value
  If check passes then assert returns object invisibly and test/check return TRUE. If check fails, assert stops code with error, check returns an error message as string, and test returns FALSE.

Examples
  testFinite(Interval$new(1, 10, class = "integer"))
  testFinite(Interval$new(1, 10, class = "numeric"))

testFuzzy

Description
  Validation checks to test if a given object is fuzzy.

Usage
  testFuzzy(object, errmsg = "This is not fuzzy.")
  checkFuzzy(object, errmsg = "This is not fuzzy.")
  assertFuzzy(object, errmsg = "This is not fuzzy.")
Arguments

- object: object to test
- errmsg: error message to overwrite default if check fails

Value

If check passes then `assert` returns object invisibly and `test/check` return `TRUE`. If check fails, `assert` stops code with error, `check` returns an error message as string, and `test` returns `FALSE`.

Examples

```r
testFuzzy(FuzzySet$new(1, 0.5))
testFuzzy(Set$new(1))
```

Description

Validation checks to test if a given object is an R6 FuzzyMultiset.

Usage

```r
testFuzzyMultiset(object, errmsg = "This is not an R6 FuzzyMultiset object")
checkFuzzyMultiset(object, errmsg = "This is not an R6 FuzzyMultiset object")
assertFuzzyMultiset(
  object,
  errmsg = "This is not an R6 FuzzyMultiset object"
)
```

Arguments

- object: object to test
- errmsg: error message to overwrite default if check fails

Value

If check passes then `assert` returns object invisibly and `test/check` return `TRUE`. If check fails, `assert` stops code with error, `check` returns an error message as string, and `test` returns `FALSE`. 
Examples

testFuzzyMultiset(Set$new(2, 3))
testFuzzyMultiset(list(Set$new(2), Set$new(3)))
testFuzzyMultiset(Tuple$new(2, 3))
testFuzzyMultiset(Interval$new())
testFuzzyMultiset(FuzzySet$new(2, 0.1))
testFuzzyMultiset(FuzzyTuple$new(2, 0.1))
testFuzzyMultiset(ConditionalSet$new(function(x) x == 0))

description:
Validation checks to test if a given object is an R6 FuzzySet.

Usage

testFuzzySet(object, errormsg = "This is not an R6 FuzzySet object")
checkFuzzySet(object, errormsg = "This is not an R6 FuzzySet object")
assertFuzzySet(object, errormsg = "This is not an R6 FuzzySet object")

Arguments

object object to test
errormsg error message to overwrite default if check fails

Value

If check passes then assert returns object invisibly and test/check return TRUE. If check fails, assert stops code with error, check returns an error message as string, and test returns FALSE.

Examples

testFuzzySet(Set$new(2, 3))
testFuzzySet(list(Set$new(2), Set$new(3)))
testFuzzySet(Tuple$new(2, 3))
testFuzzySet(Interval$new())
testFuzzySet(FuzzySet$new(2, 0.1))
testFuzzySet(FuzzyTuple$new(2, 0.1))
testFuzzySet(ConditionalSet$new(function(x) x == 0))
testFuzzyTuple  

Description
Validation checks to test if a given object is an R6 FuzzyTuple.

Usage

```r
testFuzzyTuple(object, errormsg = "This is not an R6 FuzzyTuple object")
```

```r
checkFuzzyTuple(object, errormsg = "This is not an R6 FuzzyTuple object")
```

```r
assertFuzzyTuple(object, errormsg = "This is not an R6 FuzzyTuple object")
```

Arguments

- **object** object to test
- **errormsg** error message to overwrite default if check fails

Value
If check passes then `assert` returns `object` invisibly and `test/check` return `TRUE`. If check fails, `assert` stops code with error, `check` returns an error message as string, and `test` returns `FALSE`.

Examples

```r
testFuzzyTuple(Set$new(2, 3))
testFuzzyTuple(list(Set$new(2), Set$new(3)))
testFuzzyTuple(Tuple$new(2, 3))
testFuzzyTuple(Interval$new())
testFuzzyTuple(FuzzySet$new(2, 0.1))
testFuzzyTuple(FuzzyTuple$new(2, 0.1))
testFuzzyTuple(ConditionalSet$new(function(x) x == 0))
```

testInterval  

Description
Validation checks to test if a given object is an R6 Interval.
Usage

```r
testInterval(object, errormsg = "This is not an R6 Interval object")
checkInterval(object, errormsg = "This is not an R6 Interval object")
assertInterval(object, errormsg = "This is not an R6 Interval object")
```

Arguments

- `object`: object to test
- `errormsg`: error message to overwrite default if check fails

Value

If check passes then `assert` returns `object` invisibly and `test/check` return `TRUE`. If check fails, `assert` stops code with error, `check` returns an error message as string, and `test` returns `FALSE`.

Examples

```r
testInterval(Set$new(2, 3))
testInterval(list(Set$new(2), Set$new(3)))
testInterval(Tuple$new(2, 3))
testInterval(Interval$new())
testInterval(FuzzySet$new(2, 0.1))
testInterval(FuzzyTuple$new(2, 0.1))
testInterval(ConditionalSet$new(function(x) x == 0))
```

description

testMultiset

Description

Validation checks to test if a given object is an R6 Multiset.

Usage

```r
testMultiset(object, errormsg = "This is not an R6 Multiset object")
checkMultiset(object, errormsg = "This is not an R6 Multiset object")
assertMultiset(object, errormsg = "This is not an R6 Multiset object")
```

Arguments

- `object`: object to test
- `errormsg`: error message to overwrite default if check fails
Value

If check passes then assert returns object invisibly and test/check return TRUE. If check fails, assert stops code with error, check returns an error message as string, and test returns FALSE.

Examples

testMultiset(Set$new(2, 3))
testMultiset(list(Set$new(2), Set$new(3)))
testMultiset(Tuple$new(2, 3))
testMultiset(Interval$new())
testMultiset(FuzzySet$new(2, 0.1))
testMultiset(FuzzyTuple$new(2, 0.1))
testMultiset(ConditionalSet$new(function(x) x == 0))

testSet(object, errmsg = "This is not an R6 Set object")

checkSet(object, errmsg = "This is not an R6 Set object")

assertSet(object, errmsg = "This is not an R6 Set object")

Arguments

<table>
<thead>
<tr>
<th>object</th>
<th>object to test</th>
</tr>
</thead>
<tbody>
<tr>
<td>errmsg</td>
<td>error message to overwrite default if check fails</td>
</tr>
</tbody>
</table>

Value

If check passes then assert returns object invisibly and test/check return TRUE. If check fails, assert stops code with error, check returns an error message as string, and test returns FALSE.

Examples

testSet(Set$new(2, 3))
testSet(list(Set$new(2), Set$new(3)))
testSet(Tuple$new(2, 3))
testSet(Interval$new())
testSet(FuzzySet$new(2, 0.1))
testSet(FuzzyTuple$new(2, 0.1))
testSet(ConditionalSet$new(function(x) x == 0))
testSetList

Description
Validation checks to test if a given object is an R6 SetList.

Usage

testSetList(object, errormsg = "One or more items in the list are not Sets")
checkSetList(object, errormsg = "One or more items in the list are not Sets")
assertSetList(object, errormsg = "One or more items in the list are not Sets")

Arguments

object object to test
errormsg error message to overwrite default if check fails

Value
If check passes then assert returns object invisibly and test/check return TRUE. If check fails, assert stops code with error, check returns an error message as string, and test returns FALSE.

Examples

testSetList(Set$new(2, 3))
testSetList(list(Set$new(2), Set$new(3)))
testSetList(Tuple$new(2, 3))
testSetList(Interval$new())
testSetList(FuzzySet$new(2, 0.1))
testSetList(FuzzyTuple$new(2, 0.1))
testSetList(ConditionalSet$new(function(x) x == 0))

testSubset

Description
Validation checks to test if given sets are subsets of a set.
Usage

testSubset(
  object,
  sets,
  proper = FALSE,
  errormsg = "sets are not subsets of the object"
)

checkSubset(
  object,
  sets,
  proper = FALSE,
  errormsg = "sets are not subsets of the object"
)

assertSubset(
  object,
  sets,
  proper = FALSE,
  errormsg = "sets are not subsets of the object"
)

Arguments

  object          object to test
  sets            sets to check
  proper          logical. If TRUE tests for proper subsets.
  errormsg        error message to overwrite default if check fails

Value

  If check passes then assert returns object invisibly and test/check return TRUE. If check fails, assert stops code with error, check returns an error message as string, and test returns FALSE.

Examples

testSubset(Set$new(1,2,3), Set$new(1,2))
testSubset(Set$new(1,2,3), Set$new(3,4))

description

Validation checks to test if a given object is an R6 Tuple.
Usage

```r
testTuple(object, errmsg = "This is not an R6 Tuple object")
```

```r
checkTuple(object, errmsg = "This is not an R6 Tuple object")
```

```r
assertTuple(object, errmsg = "This is not an R6 Tuple object")
```

Arguments

- **object**: object to test
- **errmsg**: error message to overwrite default if check fails

Value

If check passes then `assert` returns `object` invisibly and `test/check` return `TRUE`. If check fails, `assert` stops code with error, `check` returns an error message as string, and `test` returns `FALSE`.

Examples

```r
testTuple(Set$new(2, 3))
testTuple(list(Set$new(2), Set$new(3)))
testTuple(Tuple$new(2, 3))
testTuple(Interval$new())
testTuple(FuzzySet$new(2, 0.1))
testTuple(FuzzyTuple$new(2, 0.1))
testTuple(ConditionalSet$new(function(x) x == 0))
```

---

**Tuple**  

*Mathematical Tuple*

Description

A general Tuple object for mathematical tuples, inheriting from `Set`.

Details

Tuples are similar to sets, except that they drop the constraint for elements to be unique, and ordering in a tuple does matter. Tuples are useful for methods including `$contains` that may require non-unique elements. They are also the return type of the product of sets. See examples.

Super class

`set6::Set` -> `Tuple`
Methods

Public methods:

- Tuple$equals()
- Tuple$isSubset()
- Tuple$clone()

Method equals(): Tests if two sets are equal.

Usage:
Tuple$equals(x, all = FALSE)

Arguments:

x  Set or vector of Sets.
all  logical. If FALSE tests each x separately. Otherwise returns TRUE only if all x pass test.

Details: An object is equal to a Tuple if it contains all the same elements, and in the same order. Infix operators can be used for:

<table>
<thead>
<tr>
<th>Equal</th>
<th>==</th>
</tr>
</thead>
<tbody>
<tr>
<td>Not equal</td>
<td>!=</td>
</tr>
</tbody>
</table>

Returns: If all is TRUE then returns TRUE if all x are equal to the Set, otherwise FALSE. If all is FALSE then returns a vector of logicals corresponding to each individual element of x.

Examples:

Tuple$new(1,2) == Tuple$new(1,2)
Tuple$new(1,2) != Tuple$new(1,2)
Tuple$new(1,1) != Set$new(1,1)

Method isSubset(): Test if one set is a (proper) subset of another

Usage:
Tuple$isSubset(x, proper = FALSE, all = FALSE)

Arguments:

x  any. Object or vector of objects to test.
proper  logical. If TRUE tests for proper subsets.
all  logical. If FALSE tests each x separately. Otherwise returns TRUE only if all x pass test.

Details: If using the method directly, and not via one of the operators then the additional boolean argument proper can be used to specify testing of subsets or proper subsets. A Set is a proper subset of another if it is fully contained by the other Set (i.e. not equal to) whereas a Set is a (non-proper) subset if it is fully contained by, or equal to, the other Set.

When calling $isSubset on objects inheriting from Interval, the method treats the interval as if it is a Set, i.e. ordering and class are ignored. Use $isSubinterval to test if one interval is a subinterval of another.

Infix operators can be used for:

| Subset          | < |
| Proper Subset  | <= |
| Superset       | > |
| Proper Superset| >= |
An object is a (proper) subset of a Tuple if it contains all (some) of the same elements, and in the same order.

Returns: If all is TRUE then returns TRUE if all x are subsets of the Set, otherwise FALSE. If all is FALSE then returns a vector of logicals corresponding to each individual element of x.

Examples:
Tuple$new(1,2,3) < Tuple$new(1,2,3,4)
Tuple$new(1,3,2) < Tuple$new(1,2,3,4)

Method clone(): The objects of this class are cloneable with this method.

Usage:
Tuple$clone(deep = FALSE)

Arguments:
deep Whether to make a deep clone.

See Also
Other sets: ConditionalSet, FuzzyMultiset, FuzzySet, FuzzyTuple, Interval, Multiset, Set

Examples
# Tuple of integers
Tuple$new(1:5)

# Tuple of multiple types
Tuple$new("a", 5, Set$new(1), Tuple$new(2))

# Each Tuple has properties and traits
t <- Tuple$new(1, 2, 3)
t$traits
t$properties

# Elements can be duplicated
Tuple$new(2, 2) != Tuple$new(2)

# Ordering does matter
Tuple$new(1, 2) != Tuple$new(2, 1)

# Method `Tuple$equals`

# Method `Tuple$isSubset`

Tuple$new(1,2,3) < Tuple$new(1,2,3,4)
Tuple$new(1,3,2) < Tuple$new(1,2,3,4)
**UnionSet**

*Set of Unions*

**Description**

UnionSet class for symbolic union of mathematical sets.

**Details**

The purpose of this class is to provide a symbolic representation for the union of sets that cannot be represented in a simpler class. Whilst this is not an abstract class, it is not recommended to construct this class directly but via the set operation methods.

**Super classes**

`set6::Set` -> `set6::SetWrapper` -> `UnionSet`

**Active bindings**

- `elements` Returns the elements in the object.
- `length` Returns the number of elements in the object.

**Methods**

**Public methods:**

- `UnionSet$new()`
- `UnionSet$strprint()`
- `UnionSet$contains()`
- `UnionSet$clone()`

**Method** `new()`: Create a new UnionSet object. It is not recommended to construct this class directly.

**Usage:**

```
UnionSet$new(setlist, lower = NULL, upper = NULL, type = NULL)
```

**Arguments:**

- `setlist` list of Sets to wrap.
- `lower` lower bound of new object.
- `upper` upper bound of new object.
- `type` closure type of new object.

**Returns:** A new UnionSet object.

**Method** `strprint()`: Creates a printable representation of the object.

**Usage:**

```
UnionSet$strprint(n = 2)
```
Arguments:
n numeric. Number of elements to display on either side of ellipsis when printing.

Returns: A character string representing the object.

Method contains(): Tests if elements x are contained in self.

Usage:
UnionSet$contains(x, all = FALSE, bound = FALSE)

Arguments:
x any. Object or vector of objects to test.
all logical. If FALSE tests each x separately. Otherwise returns TRUE only if all x pass test.
bound logical.

Returns: If all == TRUE then returns TRUE if all x are contained in self, otherwise FALSE. If all == FALSE returns a vector of logicals corresponding to the length of x, representing if each is contained in self. If bound == TRUE then an element is contained in self if it is on or within the (possibly-open) bounds of self, otherwise TRUE only if the element is within self or the bounds are closed.

Method clone(): The objects of this class are cloneable with this method.

Usage:
UnionSet$clone(deep = FALSE)

Arguments:
deep Whether to make a deep clone.

See Also
Set operations: setunion, setproduct, setpower, setcomplement, setsymdiff, powerset, setintersect
Other wrappers: ComplementSet, ExponentSet, PowersetSet, ProductSet

---

Universal Mathematical Universal Set

Description

The Universal is defined as the Set containing all possible elements.

Details

The Universal set is the default universe to all sets, and is the largest possible set. The Universal set contains every single possible element. We denote the Universal set with \( V \) instead of \( U \) to avoid confusion with the union symbol. The Universal set cardinality is set to \( \text{Inf} \) where we assume \( \text{Inf} \) is greater than any \( \text{Aleph} \) or \( \text{Beth} \) numbers. The Universal set is also responsible for a few set paradoxes, to resolve these we use the following results:

Let \( V \) be the universal set, \( S \) be any non-universal set, and \( 0 \) the empty set, then
\[ V \cup S = V \]
\[ V \cap S = S \]
\[ S - V = 0 \]
\[ V^{\infty} = V \]
\[ P(V) = V \]

Super class

\texttt{set6::Set} -\texttt{Universal}

Methods

Public methods:
- \texttt{Universal\$new()}
- \texttt{Universal\$equals()}
- \texttt{Universal\$isSubset()}
- \texttt{Universal\$contains()}
- \texttt{Universal\$strprint()}
- \texttt{Universal\$clone()}

Method \texttt{new()}: Create a new \texttt{Universal} object.

Usage:
\texttt{Universal\$new()}

Details: The Universal set is the set containing every possible element.

Returns: A new Universal object.

Method \texttt{equals()}: Tests if two sets are equal.

Usage:
\texttt{Universal\$equals(x, all = FALSE)}

Arguments:
x Set or vector of Sets.
all logical. If FALSE tests each x separately. Otherwise returns TRUE only if all x pass test.

Returns: If all is TRUE then returns TRUE if all x are equal to the Set, otherwise FALSE. If all is FALSE then returns a vector of logicals corresponding to each individual element of x.
Infix operators can be used for:

\begin{verbatim}
  Equal     ==
  Not equal !=
\end{verbatim}

Examples:
# Equals
Set$new(1,2)$equals(Set$new(5,6))
Set$new(1,2)$equals(Interval$new(1,2))
Set$new(1,2) == Interval$new(1,2, class = "integer")

# Not equal
!Set$new(1,2)$equals(Set$new(1,2))
Set$new(1,2) != Set$new(1,5)

Method isSubset(): Test if one set is a (proper) subset of another

Usage:
Universal$isSubset(x, proper = FALSE, all = FALSE)

Arguments:
x any. Object or vector of objects to test.
proper logical. If TRUE tests for proper subsets.
all logical. If FALSE tests each x separately. Otherwise returns TRUE only if all x pass test.

Details: If using the method directly, and not via one of the operators then the additional boolean argument proper can be used to specify testing of subsets or proper subsets. A Set is a proper subset of another if it is fully contained by the other Set (i.e. not equal to) whereas a Set is a (non-proper) subset if it is fully contained by, or equal to, the other Set.
When calling isSubset on objects inheriting from Interval, the method treats the interval as if it is a Set, i.e. ordering and class are ignored. Use isSubinterval to test if one interval is a subinterval of another.

Infix operators can be used for:

Subset <
Proper Subset <=
Superset >
Proper Superset >=

Every Set is a subset of a Universal. No Set is a super set of a Universal, and only a Universal is not a proper subset of a Universal.

Returns: If all is TRUE then returns TRUE if all x are subsets of the Set, otherwise FALSE. If all is FALSE then returns a vector of logics corresponding to each individual element of x.

Examples:
Set$new(1,2,3)$isSubset(Set$new(1,2), proper = TRUE)
Set$new(1,2) < Set$new(1,2,3) # proper subset

c(Set$new(1,2,3), Set$new(1)) < Set$new(1,2,3) # not proper
Set$new(1,2,3) <= Set$new(1,2,3) # proper

Method contains(): Tests to see if x is contained in the Set.

Usage:
Universal$contains(x, all = FALSE, bound = NULL)

Arguments:
x any. Object or vector of objects to test.
all logical. If FALSE tests each x separately. Otherwise returns TRUE only if all x pass test.
bound ignored.

Details: x can be of any type, including a Set itself. x should be a tuple if checking to see if it lies within a set of dimension greater than one. To test for multiple x at the same time, then provide these as a list.
If using the method directly, and not via one of the operators then the additional boolean arguments all and bound. If all = TRUE then returns TRUE if all x are contained in the Set, otherwise returns a vector of logicals. For Intervals, bound is used to specify if elements lying on the (possibly open) boundary of the interval are considered contained (bound = TRUE) or not (bound = FALSE).

Returns: If all is TRUE then returns TRUE if all elements of x are contained in the Set, otherwise FALSE. If all is FALSE then returns a vector of logicals corresponding to each individual element of x.
The infix operator %\text{inset}% is available to test if x is an element in the Set, see examples.
Every element is contained within the Universal set.

Examples:
s = Set$new(1:5)

# Simplest case
s$contains(4)
8 %\text{inset}% s

# Test if multiple elements lie in the set
s$contains(4:6, all = FALSE)
s$contains(4:6, all = TRUE)

# Check if a tuple lies in a Set of higher dimension
s2 = s * s
s2$contains(Tuple$new(2,1))
c(Tuple$new(2,1), Tuple$new(1,7), 2) %\text{inset}% s2

Method strprint(): Creates a printable representation of the object.

Usage:
Universal$strprint(n = NULL)

Arguments:
n numeric. Number of elements to display on either side of ellipsis when printing.

Returns: A character string representing the object.

Method clone(): The objects of this class are cloneable with this method.

Usage:
Universal$clone(deep = FALSE)

Arguments:
deepr Whether to make a deep clone.
See Also

Other special sets: Complex, ExtendedReals, Integers, Logicals, Naturals, NegIntegers, NegRationals, NegReals, PosIntegers, PosNaturals, PosRationals, PosReals, Rationals, Reals

Examples

```r
u <- Universal$new()
print(u)
u$contains(c(1, letters, TRUE, Set$new()), all = TRUE)

# -------------------------------
# Method `Universal$equals`
# -------------------------------

# Equals
Set$new(1,2)$equals(Set$new(5,6))
Set$new(1,2)$equals(Interval$new(1,2))
Set$new(1,2) == Interval$new(1,2, class = "integer")

# Not equal
!Set$new(1,2)$equals(Set$new(1,2))
Set$new(1,2) != Set$new(1,5)

# -------------------------------
# Method `Universal$isSubset`
# -------------------------------
Set$new(1,2,3)$isSubset(Set$new(1,2), proper = TRUE)
Set$new(1,2) < Set$new(1,2,3) # proper subset

# Check if multiple elements lie in the set
s$contains(4)
s %inset% s

# Test if multiple elements lie in the set
s$contains(4:6, all = FALSE)
s$contains(4:6, all = TRUE)

# Check if a tuple lies in a Set of higher dimension
s2 = s * s
s2$contains(Tuple$new(2,1))
c(Tuple$new(2,1), Tuple$new(1,7), 2) %inset% s2
```
Description

The UniversalSet is defined as the Set containing all possible elements.

Details

The Universal set is the default universe to all sets, and is the largest possible set. The Universal set contains every single possible element. We denote the Universal set with \( V \) instead of \( U \) to avoid confusion with the union symbol. The Universal set cardinality is set to \( \text{Inf} \) where we assume \( \text{Inf} \) is greater than any Aleph or Beth numbers. The Universal set is also responsible for a few set paradoxes, to resolve these we use the following results:

Let \( V \) be the universal set, \( S \) be any non-universal set, and \( 0 \) the empty set, then

\[
\begin{align*}
V \cup S &= V \\
V \cap S &= S \\
S - V &= 0 \\
V^n &= V \\
P(V) &= V
\end{align*}
\]

Super class

set6::Set -> UniversalSet

Methods

Public methods:

- UniversalSet$new()
- UniversalSet$equals()
- UniversalSet$isSubset()
- UniversalSet$contains()
- UniversalSet$strprint()
- UniversalSet$clone()

Method new(): Create a new UniversalSet object.

Usage:

\`
UniversalSet$new()
``

Details: The Universal set is the set containing every possible element.

Returns: A new UniversalSet object.

Method equals(): Tests if two sets are equal.
Usage:
`UniversalSet$equals(x, all = FALSE)`

Arguments:
- `x` Set or vector of Sets.
- `all` logical. If FALSE tests each x separately. Otherwise returns TRUE only if all x pass test.

Returns: If all is TRUE then returns TRUE if all x are equal to the Set, otherwise FALSE. If all is FALSE then returns a vector of logicals corresponding to each individual element of x. Infix operators can be used for:

```
Equal   ==
Not equal  !=
```

Examples:
- # Equals
  ```
  Set$new(1,2)$equals(Set$new(5,6))
  Set$new(1,2)$equals(Interval$new(1,2))
  Set$new(1,2) == Interval$new(1,2, class = "integer")
  ```
- # Not equal
  ```
  !Set$new(1,2)$equals(Set$new(1,2))
  Set$new(1,2) != Set$new(1,5)
  ```

Method `isSubset()`: Test if one set is a (proper) subset of another

Usage:
`UniversalSet$isSubset(x, proper = FALSE, all = FALSE)`

Arguments:
- `x` any. Object or vector of objects to test.
- `proper` logical. If TRUE tests for proper subsets.
- `all` logical. If FALSE tests each x separately. Otherwise returns TRUE only if all x pass test.

Details: If using the method directly, and not via one of the operators then the additional boolean argument `proper` can be used to specify testing of subsets or proper subsets. A Set is a proper subset of another if it is fully contained by the other Set (i.e. not equal to) whereas a Set is a (non-proper) subset if it is fully contained by, or equal to, the other Set. When calling `isSubset` on objects inheriting from `Interval`, the method treats the interval as if it is a Set, i.e. ordering and class are ignored. Use `isSubinterval` to test if one interval is a subinterval of another. Infix operators can be used for:

```
Subset   <
Proper Subset  <=
Superset      >
Proper Superset >=
```

Every Set is a subset of a UniversalSet. No Set is a super set of a UniversalSet, and only a
UniversalSet is not a proper subset of a UniversalSet.

**Returns**: If all is TRUE then returns TRUE if all x are subsets of the Set, otherwise FALSE. If all is FALSE then returns a vector of logicals corresponding to each individual element of x.

**Examples**:

```r
Set$new(1,2,3)$isSubset(Set$new(1,2), proper = TRUE)
Set$new(1,2) < Set$new(1,2,3) # proper subset

c(Set$new(1,2,3), Set$new(1)) < Set$new(1,2,3) # not proper
Set$new(1,2,3) <= Set$new(1,2,3) # proper
```

**Method** contains(): Tests to see if x is contained in the Set.

**Usage**:

```r
UniversalSet$contains(x, all = FALSE, bound = NULL)
```

**Arguments**:

- `x` any. Object or vector of objects to test.
- `all` logical. If FALSE tests each x separately. Otherwise returns TRUE only if all x pass test.
- `bound` ignored.

**Details**: x can be of any type, including a Set itself. x should be a tuple if checking to see if it lies within a set of dimension greater than one. To test for multiple x at the same time, then provide these as a list.

If using the method directly, and not via one of the operators then the additional boolean arguments all and bound. If all = TRUE then returns TRUE if all x are contained in the Set, otherwise returns a vector of logicals. For Intervals, bound is used to specify if elements lying on the (possibly open) boundary of the interval are considered contained (bound = TRUE) or not (bound = FALSE).

**Returns**: If all is TRUE then returns TRUE if all elements of x are contained in the Set, otherwise FALSE. If all is FALSE then returns a vector of logicals corresponding to each individual element of x.

The infix operator %inset% is available to test if x is an element in the Set, see examples.

Every element is contained within the Universal set.

**Examples**:

```r
s = Set$new(1:5)

# Simplest case
s$contains(4)
8 %inset% s

# Test if multiple elements lie in the set
s$contains(4:6, all = FALSE)
s$contains(4:6, all = TRUE)

# Check if a tuple lies in a Set of higher dimension
s2 = s * s
s2$contains(Tuple$new(2,1))
c(Tuple$new(2,1), Tuple$new(1,7), 2) %inset% s2
```
Method `strprint()`: Creates a printable representation of the object.

Usage:
UniversalSet$strprint(n = NULL)

Arguments:
- `n` numeric. Number of elements to display on either side of ellipsis when printing.

Returns: A character string representing the object.

Method `clone()`: The objects of this class are cloneable with this method.

Usage:
UniversalSet$clone(deep = FALSE)

Arguments:
- `deep` Whether to make a deep clone.

Examples

```r
t <- UniversalSet$new()
print(t)
t$contains(c(1, letters, TRUE, Set$new()), all = TRUE)
```

```
# -------------------------------------
# Method `UniversalSet$equals`
# -------------------------------------

# Equals
Set$new(1,2)$equals(Set$new(5,6))
Set$new(1,2)$equals(Interval$new(1,2))
Set$new(1,2) == Interval$new(1,2, class = "integer")

# Not equal
!Set$new(1,2)$equals(Set$new(1,2))
Set$new(1,2) != Set$new(1,5)

# -------------------------------------
# Method `UniversalSet$isSubset`
# -------------------------------------

Set$new(1,2,3)$isSubset(Set$new(1,2), proper = TRUE)
Set$new(1,2) < Set$new(1,2,3) # proper subset

c(Set$new(1,2,3), Set$new(1)) < Set$new(1,2,3) # not proper
Set$new(1,2,3) <= Set$new(1,2,3) # proper

# -------------------------------------
# Method `UniversalSet$contains`
# -------------------------------------

s = Set$new(1:5)

# Simplest case
```
s$contains(4)
8 %inset% s

# Test if multiple elements lie in the set
s$contains(4:6, all = FALSE)
s$contains(4:6, all = TRUE)

# Check if a tuple lies in a Set of higher dimension
s2 = s * s
s2$contains(Tuple$new(2,1))
c(Tuple$new(2,1), Tuple$new(1,7), 2) %inset% s2

---

useUnicode  Get/Set Unicode Printing Method

Description
Change whether unicode symbols should be used when printing sets.

Usage
useUnicode(use)

Arguments
use  logical, if TRUE unicode will be used in printing, otherwise simpler character strings. If missing the current setting is returned.

Details
Using unicode symbols makes the printing of sets and properties 'prettier', however may not work on all machines or versions of R. Therefore this function is used to decide whether unicode representations should be used, or standard alpha-numeric and special characters.
By default set6 starts with unicode printing turned on.

Examples
current <- useUnicode()
useUnicode(TRUE)
useUnicode()
useUnicode(current)
Index

!=.Set (equals), 19

* coercions
  as.FuzzySet, 4
  as.Interval, 7
  as.Set, 8

* operators
  powerset, 52
  setcomplement, 68
  setintersect, 70
  setpower, 72
  setproduct, 74
  setsymdiff, 75
  setunion, 76

* sets
  ConditionalSet, 15
  FuzzyMultiset, 22
  FuzzySet, 25
  FuzzyTuple, 30
  Interval, 34
  Multiset, 42
  Set, 60
  Tuple, 95

* special sets
  Complex, 12
  ExtendedReals, 21
  Integers, 33
  Logicals, 40
  Naturals, 44
  NegIntegers, 45
  NegRationals, 46
  NegReals, 47
  PosIntegers, 48
  PosNaturals, 49
  PosRationals, 50
  PosReals, 51
  Rationals, 58
  Reals, 59
  Universal, 99

* wrappers
  ComplementSet, 10
  ExponentSet, 20
  PowersetSet, 53
  ProductSet, 55
  UnionSet, 98
  *.Set (setproduct), 74
  +.Set (setunion), 76
  -.Set (setcomplement), 68
  <=.Set (isSubset), 39
  <=.Set (isSubset), 39
  ==.Set (equals), 19
  >.Set (isSubset), 39
  >=.Set (isSubset), 39
  %-%(setsymdiff), 75
  %inset%(contains), 19
  &.Set (setintersect), 70
  ^.Set (setpower), 72

  as.FuzzyMultiset (as.FuzzySet), 4
  as.FuzzySet, 4, 8, 10
  as.FuzzySet.Set, 6
  as.FuzzyTuple (as.FuzzySet), 4
  as.Interval, 6, 7, 10
  as.Interval.numeric, 7
  as.Interval.Set, 8
  as.Multiset (as.Set), 8
  as.Set, 6, 8, 8
  as.Set.Interval, 6
  as.Tuple (as.Set), 8
  assertClosed (testClosed), 81
  assertClosedAbove (testClosedAbove), 82
  assertClosedBelow (testClosedBelow), 82
  assertConditionalSet
    (testConditionalSet), 83
  assertContains (testContains), 84
  assertCountablyFinite
    (testCountablyFinite), 85
  assertCrisp (testCrisp), 85
  assertEmpty (testEmpty), 86
  assertFinite (testFinite), 87
assertFuzzy (testFuzzy), 87
assertFuzzyMultiset
 (testFuzzyMultiset), 88
assertFuzzySet (testFuzzySet), 89
assertFuzzyTuple (testFuzzyTuple), 90
assertInterval (testInterval), 90
assertMultiset (testMultiset), 91
assertSet (testSet), 92
assertSetList (testSetList), 93
assertSubset (testSubset), 93
assertTuple (testTuple), 94
checkClosed (testClosed), 81
checkClosedAbove (testClosedAbove), 82
checkClosedBelow (testClosedBelow), 82
checkConditionalSet
 (testConditionalSet), 83
checkContains (testContains), 84
checkCountablyFinite
 (testCountablyFinite), 85
checkCrisp (testCrisp), 85
checkEmpty (testEmpty), 86
checkFinite (testFinite), 87
checkFuzzy (testFuzzy), 87
checkFuzzyMultiset (testFuzzyMultiset)
 , 88
checkFuzzySet (testFuzzySet), 89
checkFuzzyTuple (testFuzzyTuple), 90
checkInterval (testInterval), 90
checkMultiset (testMultiset), 91
checkSet (testSet), 92
checkSetList (testSetList), 93
checkSubset (testSubset), 93
checkTuple (testTuple), 94
ComplementSet, 10, 21, 54, 56, 65, 99
Complex, 12, 22, 34, 41, 45–51, 59, 60, 103
ConditionalSet, 15, 24, 29, 32, 37, 43, 66, 69, 71, 77, 97
contains, 19
equals, 19
ExponentSet, 12, 20, 54, 56, 99
ExtendedReals, 15, 21, 34, 41, 45–51, 59, 60, 103
FuzzyMultiset, 18, 22, 24, 28, 29, 32, 37, 43, 66, 97
FuzzySet, 4, 6, 8, 10, 18, 24, 25, 27, 28, 31, 32, 37, 43, 60, 66, 97
FuzzyTuple, 4, 6, 18, 24, 25, 28, 29, 30, 32, 37, 43, 66, 97
Integers, 15, 22, 33, 41, 45–51, 59, 60, 103
Interval, 7, 8, 10, 13, 14, 18, 24, 29, 32, 34, 36, 37, 43, 60, 65, 66, 96, 97, 101, 102, 105, 106
isSubset, 39
listSpecialSets, 40, 80
Logicals, 15, 22, 34, 40, 45–51, 59, 60, 103
LogicalSet, 41
Multiset, 18, 22, 24, 29, 32, 37, 42, 66, 97
Naturals, 15, 22, 34, 41, 44, 46–51, 59, 60, 103
NegIntegers, 15, 22, 34, 41, 45, 47–51, 59, 60, 103
NegRationals, 15, 22, 34, 41, 45, 46, 48–51, 59, 60, 103
NegReals, 15, 22, 34, 41, 45–47, 47, 48–51, 59, 60, 103
PosIntegers, 15, 22, 34, 41, 45–48, 48, 49–51, 59, 60, 103
PosNaturals, 15, 22, 34, 41, 45–48, 49, 50, 51, 59, 60, 103
PosRationals, 15, 22, 34, 41, 45–49, 50, 51, 59, 60, 103
PosReals, 15, 22, 34, 41, 45–50, 51, 59, 60, 103
powerset, 12, 21, 52, 54, 56, 70, 71, 73, 74, 76, 77, 99
PowerSetSet, 12, 21, 52, 53, 56, 99
ProductSet, 12, 21, 54, 55, 74, 99
Properties, 57
Rationals, 15, 22, 34, 41, 45–51, 58, 60, 103
Reals, 15, 22, 34, 35, 41, 45–51, 59, 59, 103
set6 (set6-package), 3
set6-package, 3
set6::FuzzySet, 22, 31
set6::Integers, 45, 48
set6::Interval, 21, 33, 44–51, 58, 60, 80
set6::Naturals, 49
<table>
<thead>
<tr>
<th>Term</th>
<th>Page Numbers</th>
</tr>
</thead>
<tbody>
<tr>
<td>set6::ProductSet</td>
<td>20, 53</td>
</tr>
<tr>
<td>set6::Rationals</td>
<td>46, 50</td>
</tr>
<tr>
<td>set6::Reals</td>
<td>21, 47, 51</td>
</tr>
<tr>
<td>set6::Set</td>
<td>10, 12, 15, 20–22, 25, 31, 33, 34, 40–42, 44–51, 53, 55, 58, 60, 78, 80, 95, 98, 100, 104</td>
</tr>
<tr>
<td>set6::SetWrapper</td>
<td>10, 20, 53, 55, 98</td>
</tr>
<tr>
<td>set6::SpecialSet</td>
<td>21, 33, 44–51, 58, 60</td>
</tr>
<tr>
<td>set6News</td>
<td>68</td>
</tr>
<tr>
<td>setcomplement</td>
<td>12, 21, 35, 52, 54, 56, 68, 71, 73, 74, 76, 77, 99</td>
</tr>
<tr>
<td>setintersect</td>
<td>12, 21, 52, 54, 56, 70, 73, 74, 76, 77, 99</td>
</tr>
<tr>
<td>setpower</td>
<td>12, 21, 52, 54, 56, 70, 71, 72, 74, 76, 77, 99</td>
</tr>
<tr>
<td>setproduct</td>
<td>12, 21, 52, 54, 56, 70, 71, 73, 74, 76, 77, 99</td>
</tr>
<tr>
<td>setsymdiff</td>
<td>12, 21, 52, 54, 56, 70, 71, 73, 74, 75, 77, 99</td>
</tr>
<tr>
<td>setunion</td>
<td>12, 21, 52, 54, 56, 70, 71, 73, 74, 76, 76, 99</td>
</tr>
<tr>
<td>SetWrapper</td>
<td>78</td>
</tr>
<tr>
<td>SpecialSet</td>
<td>80</td>
</tr>
<tr>
<td>testClosed</td>
<td>81</td>
</tr>
<tr>
<td>testClosedAbove</td>
<td>82</td>
</tr>
<tr>
<td>testClosedBelow</td>
<td>82</td>
</tr>
<tr>
<td>testConditionalSet</td>
<td>83</td>
</tr>
<tr>
<td>testContains</td>
<td>84</td>
</tr>
<tr>
<td>testCountablyFinite</td>
<td>85</td>
</tr>
<tr>
<td>testCrisp</td>
<td>85</td>
</tr>
<tr>
<td>testEmpty</td>
<td>86</td>
</tr>
<tr>
<td>testFinite</td>
<td>87</td>
</tr>
<tr>
<td>testFuzzy</td>
<td>87</td>
</tr>
<tr>
<td>testFuzzyMultiset</td>
<td>88</td>
</tr>
<tr>
<td>testFuzzySet</td>
<td>89</td>
</tr>
<tr>
<td>testFuzzyTuple</td>
<td>90</td>
</tr>
<tr>
<td>testInterval</td>
<td>90</td>
</tr>
<tr>
<td>testMultiset</td>
<td>91</td>
</tr>
<tr>
<td>testSet</td>
<td>92</td>
</tr>
<tr>
<td>testSetList</td>
<td>93</td>
</tr>
<tr>
<td>testSubset</td>
<td>93</td>
</tr>
<tr>
<td>testTuple</td>
<td>94</td>
</tr>
<tr>
<td>Tuple</td>
<td>8, 10, 16, 18, 24, 29, 31, 32, 37, 42, 43, 60, 66, 95</td>
</tr>
<tr>
<td>UniversalSet</td>
<td>104</td>
</tr>
<tr>
<td>useUnicode</td>
<td>108</td>
</tr>
<tr>
<td>useUnicode()</td>
<td>62</td>
</tr>
</tbody>
</table>