

Package ‘container’

July 22, 2025

Type Package

Title Extending Base 'R' Lists

Version 1.0.5

Description Extends the functionality of base 'R' lists and provides specialized data structures 'deque', 'set', 'dict', and 'dict.table', the latter to extend the 'data.table' package.

Depends R (>= 3.5.0)

License GPL-3

Encoding UTF-8

Imports data.table, methods, R6

VignetteBuilder knitr

Suggests knitr, tinytest, rmarkdown, microbenchmark, ggplot2, dplyr, tibble, testthat (>= 3.0.0)

URL <https://rpahl.github.io/container/>,
<https://github.com/rpahl/container>

BugReports <https://github.com/rpahl/container/issues>

NeedsCompilation no

Author Roman Pahl [aut, cre]

Maintainer Roman Pahl <roman.pahl@gmail.com>

Repository CRAN

RoxygenNote 7.2.1

Config/testthat/edition 3

Date/Publication 2024-12-02 20:00:02 UTC

Contents

add	3
addleft	4
at	5
at2	6
clear	7
clone	8
Container	9
ContainerS3	16
container_options	22
count	24
delete	25
delete_at	25
deprecated	27
Deque	28
DequeS3	30
Dict	32
dict.table	36
DictS3	41
discard	43
discard_at	44
has	45
has_name	46
is_empty	47
Iterable	48
Iterator	49
iterS3	51
OpsArithmetic	53
OpsCompare	55
OpsExtract	56
OpsLogic	57
OpsReplace	58
OrderedSet	59
peek	61
peek_at	62
peek_at2	63
pop	64
rename	66
replace	67
replace_at	68
rev	70
rotate	71
Set	72
SetS3	74
unpack	76
update	77

add	<i>Add Elements to Containers</i>
-----	-----------------------------------

Description

Add elements to container-like objects.

Usage

```
add(.x, ...)  
  
ref_add(.x, ...)  
  
## S3 method for class 'Container'  
add(.x, ...)  
  
## S3 method for class 'Container'  
ref_add(.x, ...)  
  
## S3 method for class 'Dict'  
add(.x, ...)  
  
## S3 method for class 'Dict'  
ref_add(.x, ...)  
  
## S3 method for class 'dict.table'  
add(.x, ...)  
  
## S3 method for class 'dict.table'  
ref_add(.x, ...)
```

Arguments

.x	an R object of the respective class.
...	elements to be added.

Value

For [Container](#), an object of class [Container](#) (or one of the respective derived classes).

For [dict.table](#) an object of class [dict.table](#).

Note

While [add](#) uses copy semantics [ref_add](#) works by reference.

If .x is a [Container](#), [Set](#) or [Deque](#) object, the elements being added can (but must not) be named.

If .x is a [Dict](#) or [dict.table](#) object, all elements *must* be of the form key = value. If one of the keys already exists, an error is given.

Examples

```

co = container(1)
add(co, 1, b = 2, c = container(1:3))

s = setnew(1)
add(s, 1, 1, b = 2, "1", co = container(1, 1))

d = dict(a = 1)
add(d, b = 2, co = container(1:3))

try(add(d, a = 7:9)) # key 'a' already in Dict

dit = dict.table(a = 1:3)
add(dit, b = 3:1, d = 4:6)

try(add(dit, a = 7:9)) # column 'a' already exists

dit = dict.table(a = 1:3)
add(dit, b = 3:1, d = 4:6)

try(add(dit, a = 7:9)) # column 'a' already exists

```

addleft

Add Elements to the Left of Deques

Description

Add elements to left side of [Deque](#) objects.

Usage

```

addleft(.x, ...)

ref_addleft(.x, ...)

## S3 method for class 'Deque'
addleft(.x, ...)

## S3 method for class 'Deque'
ref_addleft(.x, ...)

```

Arguments

.x	a [Deque] object
...	elements to be added.

Value

For [Deque](#), an object of class [Deque](#) with the elements being added to the left of .x.

Note

While `addleft` uses copy semantics `ref_addleft` work by reference.

Examples

```
d = deque(0)
add(d, a = 1, b = 2)      # |0, a = 1, b = 2|
addleft(d, a = 1, b = 2)  # |b = 2, a = 1, 0|
```

at

Extract Elements Safely

Description

Extract parts of a Container at given indices. If an index is invalid, an error is signaled. If given as a string, the element matching the name is returned. If there are two or more identical names, the value of the first match (i.e. *leftmost* element) is returned. Indices can be letters or numbers, or both at the same time.

Usage

```
at(.x, ...)

## S3 method for class 'Container'
at(.x, ...)

## S3 method for class 'dict.table'
at(.x, ...)
```

Arguments

```
.x          an R object of the respective class.
...         indices of elements to be extracted
```

Value

For `Container`, returns the values at the given indices.

For `dict.table`, returns the columns at the given indices.

See Also

`peek_at()` for less strict extraction

Examples

```
# Container
co = container(a = 1, 2, b = 3, 4)
at(co, 1:3)
at(co, "a", "b", 2)
try(at(co, "x"))      # index 'x' not found
try(at(co, 1:10))     # index 5 exceeds length of Container
# Dict
d = dict(a = 1, b = 3)
at(d, 1:2)
at(d, "a", 2)
try(at(d, "x"))       # index 'x' not found
try(at(d, 1:3))       # index 5 exceeds length of Dict

# dict.table
dit = dict.table(a = 1:3, b = 4:6)
at(dit, "a")
at(dit, 2)
at(dit, "a", 2)
try(at(dit, "x"))     # index 'x' not found
try(at(dit, 1:3))     # index 3 exceeds length of dict.table
```

at2

Extract Single Elements Safely

Description

Extracts the value of a Container at the given index. If the index is invalid, an error is signaled. If given as a string, the element matching the name is returned. If there are two or more identical names, the value of the first match (i.e. *leftmost* element) is returned. Extract value at index. If index is invalid or not found, an error is signaled. If given as a string, the element matching the name is returned. If there are two or more identical names, the value of the first match (i.e. *leftmost* element) is returned.

Usage

```
at2(x, ...)

## S3 method for class 'Container'
at2(x, index, ...)

## S3 method for class 'dict.table'
at2(x, index, ...)
```

Arguments

x	an R object of the respective class.
...	other arguments passed to or from methods.
index	character name or numeric position of the sought value.

Value

For `Container`, returns the value at the given index.

For `dict.table`, returns the column at the given index or signals an error if not found.

See Also

[peek_at2\(\)](#) for less strict extraction

Examples

```
# Container
co = container(a = 1, 2, b = 3, 4)
at2(co, 1)
at2(co, "a")
at2(co, 2)
try(at2(co, "x"))      # index 'x' not found
try(at2(co, 5))        # index 5 exceeds length of Container

# Dict
d = dict(a = 1, b = 3)
at2(d, 1)
at2(d, "a")
at2(d, 2)
try(at2(d, "x"))      # index 'x' not found
try(at2(d, 5))        # index 5 exceeds length of Dict

# dict.table
dit = dict.table(a = 1:3, b = 4:6)
at2(dit, 1)
at2(dit, "a")
at2(dit, 2)
try(at2(dit, "x"))    # index 'x' not found
try(at2(dit, 5))      # index 5 exceeds length of dict.table
```

clear

Clear a Container

Description

Removes all elements from the container object.

Usage

```
clear(x)
```

```
ref_clear(x)
```

```
## S3 method for class 'Container'
clear(x)
```

```
## S3 method for class 'Container'
ref_clear(x)

## S3 method for class 'dict.table'
clear(x)

## S3 method for class 'dict.table'
ref_clear(x)
```

Arguments

x any R object.

Value

For [Container](#), an object of class [Container](#) (or one of the respective derived classes).

For [dict.table](#) an object of class [dict.table](#).

Examples

```
co = container(1, 2, mean)
clear(co)
co
ref_clear(co)
co

dit = dict.table(a = 1, b = 2)
clear(dit)
dit          # original was not touched
ref_clear(dit)
dit          # original was cleared
```

clone

Clone an Object

Description

Creates a copy of the object.

Usage

```
clone(x)

## S3 method for class 'Container'
clone(x)

## S3 method for class 'dict.table'
clone(x)
```


Arguments

x any R object.

Value

A copy of the object.

Examples

```
co = container(1, 2, 3)
co2 = clone(co)
co == co2

d = dict.table(a = 1:2, b = 3:4)
d2 = clone(d)
ref_clear(d)
print(d2)
```

Container

Container Class

Description

This class implements a container data structure with typical member functions to insert, delete and access elements from the container. For the standard S3 interface, see [container\(\)](#).

Details

This class inherits from class [Iterable](#) and serves as the base class for [Deque](#), [Set](#), and [Dict](#).

Super class

[container::Iterable](#) -> Container

Methods**Public methods:**

- [Container\\$new\(\)](#)
- [Container\\$add\(\)](#)
- [Container\\$at\(\)](#)
- [Container\\$at2\(\)](#)
- [Container\\$clear\(\)](#)
- [Container\\$count\(\)](#)
- [Container\\$delete\(\)](#)
- [Container\\$delete_at\(\)](#)
- [Container\\$discard\(\)](#)

- `Container$discard_at()`
- `Container$empty()`
- `Container$get_compare_fun()`
- `Container$has()`
- `Container$has_name()`
- `Container$is_empty()`
- `Container$length()`
- `Container$names()`
- `Container$peek_at()`
- `Container$peek_at2()`
- `Container$pop()`
- `Container$print()`
- `Container$rename()`
- `Container$replace()`
- `Container$replace_at()`
- `Container$remove()`
- `Container$size()`
- `Container$type()`
- `Container$update()`
- `Container$values()`
- `Container$clone()`

Method `new()`: constructor

Usage:

`Container$new(...)`

Arguments:

... initial elements put into the Container

Returns: the Container object

Method `add()`: add element

Usage:

`Container$add(value, name = NULL)`

Arguments:

value value of ANY type to be added to the Container.

name character optional name attribute of the value.

Returns: the Container object

Method `at()`: Same as `at2` (see below) but accepts a vector of indices and always returns a Container object.

Usage:

`Container$at(index)`

Arguments:

index vector of indices.

Returns: Container object with the extracted elements.

Method `at2()`: Extract value at index. If index is invalid or not found, an error is signaled. If given as a string, the element matching the name is returned. If there are two or more identical names, the value of the first match (i.e. *leftmost* element) is returned.

Usage:

`Container$at2(index)`

Arguments:

index Must be a single number > 0 or a string.

Returns: If given as a number, the element at the corresponding position, and if given as a string, the element at the corresponding name matching the given string is returned.

Method `clear()`: delete all elements from the Container

Usage:

`Container$clear()`

Returns: the cleared Container object

Method `count()`: Count number of element occurrences.

Usage:

`Container$count(elem)`

Arguments:

elem element to be counted.

Returns: integer number of elem occurrences in the [Container\(\)](#)

Method `delete()`: Search for occurrence(s) of elem in Container and remove first one that is found. If elem does not exist, an error is signaled.

Usage:

`Container$delete(elem)`

Arguments:

elem element to be removed from the Container.

Returns: the Container object

Method `delete_at()`: Delete value at given index. If index is not found, an error is signaled.

Usage:

`Container$delete_at(index)`

Arguments:

index character or numeric index

Returns: the Container object

Method `discard()`: Search for occurrence(s) of elem in Container and remove first one that is found.

Usage:

Container\$discard(elem)

Arguments:

elem element to be discarded from the Container. If not found, the operation is ignored and the object is *not* altered.

Returns: the Container object

Method discard_at(): Discard value at given index. If index is not found, the operation is ignored.

Usage:

Container\$discard_at(index)

Arguments:

index character or numeric index

Returns: the Container object

Method empty(): This function is deprecated. Use [is_empty\(\)](#) instead.

Usage:

Container\$empty()

Method get_compare_fun(): Get comparison function used internally by the Container object to compare elements.

Usage:

Container\$get_compare_fun()

Method has(): Determine if Container has some element.

Usage:

Container\$has(elem)

Arguments:

elem element to search for

Returns: TRUE if Container contains elem else FALSE

Method has_name(): Determine if Container object contains an element with the given name. If called with no argument, the function determines whether *any* element is named.

Usage:

Container\$has_name(name)

Arguments:

name character the name

Returns: TRUE if Container has the name otherwise FALSE

Method is_empty(): Check if Container is empty

Usage:

Container\$is_empty()

Returns: TRUE if the Container is empty else FALSE.

Method length(): Number of elements of the Container.

Usage:

Container\$length()

Returns: integer length of the Container, that is, the number of elements it contains.

Method names(): Names of the elements.

Usage:

Container\$names()

Returns: character the names of the elements contained in x

Method peek_at(): Same as peek_at2 (see below) but accepts a vector of indices and always returns a Container object.

Usage:

Container\$peek_at(index, default = NULL)

Arguments:

index vector of indices.

default the default value to return in case the value at index is not found.

Returns: Container object with the extracted elements.

Method peek_at2(): Peek at index and extract value. If index is invalid, missing, or not not found, return default value.

Usage:

Container\$peek_at2(index, default = NULL)

Arguments:

index numeric or character index to be accessed.

default the default value to return in case the value at index is not found.

Returns: the value at the given index or (if not found) the given default value.

Method pop(): Get value at index and remove it from Container. If index is not found, raise an error.

Usage:

Container\$pop(index)

Arguments:

index Must be a single number > 0 or a string.

Returns: If given as a number, the element at the corresponding position, and if given as a string, the element at the corresponding name matching the given string is returned.

Method print(): Print object representation

Usage:

Container\$print(...)

Arguments:

... further arguments passed to `format()`

Returns: invisibly returns the Container object

Method `rename()`: Rename a key in the Container. An error is signaled, if either the old key is not in the Container or the new key results in a name-clash with an existing key.

Usage:

```
Container$rename(old, new)
```

Arguments:

old character name of key to be renamed.

new character new key name.

Returns: the Container object

Method `replace()`: Replace one element by another element. Search for occurrence of old and, if found, replace it by new. If old does not exist, an error is signaled, unless add was set to TRUE, in which case new is added.

Usage:

```
Container$replace(old, new, add = FALSE)
```

Arguments:

old element to be replaced

new element to be put instead of old

add logical if TRUE the new element is added in case old does not exist.

Returns: the Container object

Method `replace_at()`: Replace value at given index. Replace value at index by given value. If index is not found, an error is signalled, unless add was set to TRUE, in which case new is added.

Usage:

```
Container$replace_at(index, value, add = FALSE)
```

Arguments:

index character or numeric index

value ANY new value to replace the old one.

add logical if TRUE the new value element would be added in case index did not exist.

Returns: the Container object

Method `remove()`: This function is deprecated. Use `delete()` instead.

Usage:

```
Container$remove(elem)
```

Arguments:

elem element to be deleted from the Container. If element is not found in the Container, an error is signaled.

Returns: the Container object

Method `size()`: This function is deprecated. Use `length()` instead.

Usage:

Container\$size()

Returns: the Container length

Method type(): This function is deprecated and of no real use anymore.

Usage:

Container\$type()

Returns: type (or mode) of internal vector containing the elements

Method update(): Add elements of other to this if the name is not in the Container and update elements with existing names.

Usage:

Container\$update(other)

Arguments:

other Iterable object used to update this.

Returns: returns the Container

Method values(): Get Container values

Usage:

Container\$values()

Returns: elements of the container as a base list

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

Usage:

Container\$clone(deep = FALSE)

Arguments:

deep Whether to make a deep clone.

Author(s)

Roman Pahl

See Also

[container\(\)](#), [Iterable](#), [Deque](#), [Set](#), and [Dict](#)

Examples

```
co = Container$new(1:5, c = Container$new("a", 1), l = list())
co$print()
co$length()
co$names()
co$clear()

# Extract
co = Container$new(a = 1, b = 2, c = 3, d = 4)
```

```

co$at(1:2)
co$at(c(1, 4))
co$at(list("d", 2))
co$at2(1)

try(co$at(0:2)) # index must be > 0

co$peek_at(0:2)
co$peek_at(0:2, default = 1)

# Replace
co$replace(4, 9)
co$replace(9, 11)
co$replace_at(1, -1)

try(co$replace_at(11, 1)) # index 11 exceeds length of Container

# Delete
co$delete(-1)
co$delete_at(3)

try(co$delete_at(3)) # index 3 exceeds length of Container

co$discard(3)

co2 = Container$new(b = 0)
co2$add(0, name = "a")
co$update(co2)
co$pop(1)
co

```

ContainerS3

Container - Enhancing R's list

Description

A container is a data structure with typical member functions to insert, delete and access elements from the container object. It can be considered as a base R [list](#) with extended functionality. The [Container](#) class also serves as the base class for [Deque](#), [Set](#), and [Dict](#) objects.

Usage

```

container(...)

cont(...)

as.container(x)

as.cont(x)

```



```

is.container(x)

## S3 method for class 'Container'
as.list(x, ...)

## S3 method for class 'Container'
length(x)

## S3 method for class 'Container'
names(x)

## S3 replacement method for class 'Container'
names(x) <- value

```

Arguments

...	(possibly named) elements to be put into or removed from the Container , or additional arguments passed from and to methods.
x	R object of ANY type for as.container and is.container or of class Container for the S3 methods.
value	character vector of names.

Details

Methods that alter [Container](#) objects usually come in two versions providing either copy or reference semantics where the latter start with 'ref_' to note the reference semantic, for example, [add\(\)](#) and [ref_add\(\)](#).

- [container\(...\)](#) initializes and returns a [Container](#) object.
- [cont\(...\)](#) is a short cut for [container\(...\)](#).
- [as.container\(x\)](#) or [as.cont\(x\)](#) coerce x to a [Container](#)
- [is.container\(x\)](#) check if x is a [Container](#)
- [as.list\(x\)](#) converts container x to a base R [list](#). All of the container's elements are copied (deeply) during the conversion.
- [length\(x\)](#) return the number of elements contained in x.
- [names\(x\)](#) return the names of the elements contained in x.
- [names\(x\) <- value](#) sets the names of x.
- $x + y$ combines x and y into a new container by appending y to x.
- $x - y$ element-wise discards all items of y from x, given the element was contained in x. The result is always a container.

- `x == y` is TRUE if the contents of `x` and `y` are lexicographically *equal*.
- `x != y` is TRUE if the contents of `x` and `y` are not equal.
- `x < y` is TRUE if the contents of `x` are lexicographically *less* than the contents of `y`.
- `x <= y` is TRUE if the contents of `x` are lexicographically *less* than or *equal* to the contents of `y`.
- `add(.x, ...)` and `ref_add(.x, ...)` add elements to `.x`.
- `at(.x, ...)` returns the value at the given indices. Indices can be letters or numbers or both. All indices must exist.
- `at2(x, index)` returns the value at the given index or signals an error if not found.
- `clear(x)` and `ref_clear(x)` remove all elements from `x`.
- `clone(x)` create a copy of `x`.
- `count(x, elem)` count how often `elem` occurs in `x`.
- `delete(.x, ...)` and `ref_delete(.x, ...)` find and remove elements. If one or more elements don't exist, an error is signaled.
- `delete_at(.x, ...)` and `ref_delete_at(.x, ...)` find and remove values at given indices. If any given index is invalid, an error is signaled.
- `discard(.x, ...)` and `ref_discard(.x, ...)` find and discard elements. Elements that don't exist, are ignored.
- `discard_at(.x, ...)` and `ref_discard_at(.x, ...)` find and discard values at given indices. Invalid indices are ignored.
- `has(x, elem)` TRUE if element is in `x` and otherwise FALSE.
- `has_name(x, name)` check if `name` is in `x`
- `is_empty(x)` TRUE if object is empty otherwise FALSE
- `peek_at(x, ..., .default = NULL)` returns the value at the given indices or (if not found) the given default value.
- `peek_at2(x, index, default)` returns the value at the given index or (if not found) the given default value.
- `ref_pop(.x, index)` return element at given index and remove it from the container object.
- `rename(.x, old, new)` and `ref_rename(.x, old, new)` rename one or more keys from `old` to `new`, respectively, by copy and in place (i.e. by reference).
- `replace(.x, old, new, add = FALSE)` and `ref_replace(.x, old, new, add = FALSE)` try to find element `old` and replace it with element `new`. If `old` does not exist, an error is raised, unless `add` was set to TRUE.
- `replace_at(.x, ..., .add = FALSE)` and `ref_replace_at(.x, ..., .add = FALSE)` replace values at given indices. If a given index is invalid, an error is signaled unless `.add` was set to TRUE.

See Also

For the class documentation see [Container](#). Objects of the derived classes can be created by [deque](#), [setnew](#), and [dict](#).

Examples

```
co = container(1:5, c = container("a", 1), l = list())
is.container(co)
print(co)
length(co)
names(co)

unpack(co)  # flatten recursively similar to unlist

# Math
co = container(1, 2, -(3:5))
co
abs(co)
cumsum(co)
round(co)
exp(co)

# Summary
range(co)
min(co)
max(co)

# Arithmetic
c1 = container(1, 1:2)
c2 = container(2, 1:2)
c1 + c2      # same as c(c1, c2)
c2 + c1      # same as c(c2, c1)

c1 - c2
c2 - c1
c1 - c1

# Comparison
c1 = container(1, 2, 3)
c2 = container(1, 3, 2)
c1 == c1      # TRUE
c1 != c2      # TRUE
c1 <= c1      # TRUE
c1 == c2      # FALSE
c1 < c2       # TRUE
c1 < container(2) # TRUE
c1 < container() # FALSE

# Extract or replace
co = container(a = 1, b = 2, c = 3, d = 4)
co[1:2]
co[1, 4]
```

```

co["d", 2]
co[list("d", 2)]
co[0:10]

co = container(a = 1, b = 2)
co[[1]]
co[["a"]]
co[["x"]]
co = container(a = 1, b = "bar")
(co[1:2] <- 1:2)

try({
  co[3] <- 3 # index out of range
})
(co[list(1, "b")] <- 3:4) # mixed numeric/character index

co = container(a = 1, b = 2)
co[[1]] <- 9
co[["b"]] <- 8
co[["x"]] <- 7
co$z <- 99
print(co)

# Replace 8 by 0
co[["b"]] <- 0
print(co)

co = container(a = 1, b = "bar")
co$f <- 3
co$b <- 2
co

co = container(1)
add(co, 1, b = 2, c = container(1:3))

co = container(a = 1, 2, b = 3, 4)
at(co, 1:3)
at(co, "a", "b", 2)
try(at(co, "x")) # index 'x' not found
try(at(co, 1:10)) # index 5 exceeds length of Container

co = container(a = 1, 2, b = 3, 4)
at2(co, 1)
at2(co, "a")
at2(co, 2)
try(at2(co, "x")) # index 'x' not found
try(at2(co, 5)) # index 5 exceeds length of Container

co = container(1, 2, mean)
clear(co)

```

```

print(co)    # Original was not touched
ref_clear(co) # Clears original
print(co)

co = container(1, 2, 3)
co2 = clone(co)
co == co2

co = container("a", "b", "a", mean, mean)
count(co, "a")
count(co, mean)
count(co, "c")

co = container("a", 1:3, iris)
print(co)
delete(co, 1:3, "a")
delete(co, iris)
try({
delete(co, "b")    # "b" is not in Container
})

co = container(a = 1, b = 2, 3)
delete_at(co, "a", "b")    # [3]
delete_at(co, 1:2)        # [3]
delete_at(co, "a", 3)      # [b = 2]
try({
  delete_at(co, 4)          # index out of range
  delete_at(co, "x")        # names(s) not found: 'x'
})

co = container("a", num = 1:3, data = iris)
print(co)
discard(co, 1:3, "a")
discard(co, iris)
discard(co, "b") # ignored

co = container(a = 1, b = 2, 3)
discard_at(co, "a", "b")    # [3]
discard_at(co, 1:2)        # [3]
discard_at(co, "a", 3)      # [b = 2]
discard_at(co, "x")        # ignored

co = container(1, 2, mean)
has(co, 1)                 # TRUE
has(co, mean)              # TRUE
has(co, 1:2)               # FALSE

co = container(a = 1, 2, f = mean)
has_name(co, "a")          # TRUE
has_name(co, "f")          # TRUE
has_name(co, "2")          # FALSE

co = container(1, 2)

```

```

is_empty(co)
is_empty(clear(co))

co = container(a = 1, 2, b = 3, 4)
peek_at(co, 1)
peek_at(co, "a")
peek_at(co, "x")
peek_at(co, "x", .default = 0)
peek_at(co, "a", "x", 2, 9, .default = -1)

co = container(a = 1, 2, b = 3, 4)
peek_at2(co, 1)
peek_at2(co, "a")
peek_at2(co, "x")
peek_at2(co, "x", default = 0)

co = container(a = 1, b = 1:3, d = "foo")
ref_pop(co, "b")
ref_pop(co, 1)

try({
  ref_pop(co, "x") # index 'x' not found
})
co = container(a = 1, b = 2, 3)
rename(co, c("a", "b"), c("a1", "y"))
print(co)
ref_rename(co, c("a", "b"), c("a1", "y"))
print(co)

co = container("x", 9)
replace(co, 9, 0)
replace(co, "x", 0)
try({
  replace(co, "z", 0) # old element ("z") is not in Container
})
replace(co, "z", 0, add = TRUE) # ok, adds the element

co = container(a = 0, b = "z")
replace_at(co, a = 1, b = 2)
replace_at(co, 1:2, 1:2) # same
replace_at(co, c("a", "b"), list(1, 2)) # same

try({
  replace_at(co, x = 1) # names(s) not found: 'x'
})
replace_at(co, x = 1, .add = TRUE) # ok (adds x = 1)

```

Description

Set Container Package Options

Usage

```
container_options(..., .reset = FALSE)
```

```
getContainerOption(x, default = NULL)
```

Arguments

<code>...</code>	any options can be defined, using <code>name = value</code> .
<code>.reset</code>	logical if TRUE, the options are reset to their default and returned.
<code>x</code>	a character string holding an option name.
<code>default</code>	if the specified option is not set in the options list, this value is returned.

Value

- `container_options()` returns a list of all set options sorted by name.
- `container_options(name)`, a list of length one containing the set value, or NULL if it is unset. Can also be multiple names (see Examples).
- `container_options(key = value)` sets the option with name `key` to `value` and returns the previous options invisibly.

Container Options

- `compare` (default = `all.equal`)
- `useDots` (default = TRUE) whether to abbreviate long container elements with `...` when exceeding `vec.len` (see below). If FALSE, they are abbreviated as `<<type(length)>>`.
- `vec.len` (default = 4) the length limit at which container vectors are abbreviated.

Examples

```
co = container(1L, 1:10, as.list(1:5))
co

container_options(useDots = FALSE)
co

container_options(useDots = TRUE, vec.len = 6)
co

has(co, 1.0)

container_options(compare = "identical")

has(co, 1.0) # still uses 'all.equal'
```

```
co2 = container(1L)
has(co2, 1.0)
has(co2, 1L)

container_options()
container_options(.reset = TRUE)
```

count

Count Elements

Description

Count the number of occurrences of some element.

Usage

```
count(x, elem)

## S3 method for class 'Container'
count(x, elem)

## S3 method for class 'Set'
count(x, elem)
```

Arguments

x	any R object.
elem	element to counted.

Value

integer number of how many times elem occurs in the object.

Examples

```
co = container("a", "b", "a", mean, mean)
count(co, "a")
count(co, mean)
count(co, "c")
```

delete	<i>Delete Container Elements Safely</i>
--------	---

Description

Search and remove elements from an object. If the element is not found, an error is signaled.

Usage

```
delete(.x, ...)

ref_delete(.x, ...)

## S3 method for class 'Container'
delete(.x, ...)

## S3 method for class 'Container'
ref_delete(.x, ...)
```

Arguments

.x	any R object.
...	elements to be deleted.

Value

For Container, an object of class Container (or one of the respective derived classes).

Examples

```
s = setnew("a", 1:3, iris)
print(s)
delete(s, 1:3, "a")
delete(s, iris)
try({
  delete(s, "b") # "b" is not in Set
})
```

delete_at	<i>Delete Elements at Indices Safely</i>
-----------	--

Description

Search and remove values at given indices, which can be numeric or character or both. If any given index is invalid, an error is signaled. Indices can be numbers or names or both.

Usage

```

delete_at(.x, ...)

ref_delete_at(.x, ...)

## S3 method for class 'Container'
delete_at(.x, ...)

## S3 method for class 'Container'
ref_delete_at(.x, ...)

## S3 method for class 'dict.table'
delete_at(.x, ...)

## S3 method for class 'dict.table'
ref_delete_at(.x, ...)

```

Arguments

```

.x                any R object.
...              indices at which values are to be deleted.

```

Value

For `Container`, an object of class `Container` (or one of the respective derived classes).

For `dict.table`, an object of class `dict.table`.

Examples

```

co = container(a = 1, b = 2, 3)
delete_at(co, "a", "b")      # [3]
delete_at(co, 1:2)          # [3]
delete_at(co, "a", 3)        # [b = 2]
try({
  delete_at(co, 4)            # index out of range
  delete_at(co, "x")          # names(s) not found: 'x'
})

dit = as.dict.table(head(sleep))
dit
delete_at(dit, "ID")
delete_at(dit, "ID", 1)
try({
  delete_at(dit, "foo")      # Column 'foo' not in dict.table
})

```

deprecated	<i>Deprecated Functions</i>
------------	-----------------------------

Description

These functions are provided for backwards-compatibility and may be defunct as soon as the next release.

Usage

```
empty(x)

## S3 method for class 'Container'
empty(x)

size(x)

## S3 method for class 'Container'
size(x)

sortkey(x, decr = FALSE)

## S3 method for class 'Dict'
sortkey(x, decr = FALSE)

values(x)

## S3 method for class 'Container'
values(x)

## S3 method for class 'dict.table'
values(x)

keys(x)
```

Arguments

x	any R object.
decr	logical sort decreasingly?

Details

- `empty()` `is_empty()` instead
- `set()` `setnew()` instead
- `size()` use `length()` instead
- `sortkey()` keys of `Dict` objects are now always sorted

- `remove()` use `delete()` instead
- `type()` not of use anymore
- `values()` use `as.list()` instead

 Deque

 Deque Class

Description

Dequeues are a generalization of stacks and queues typically with methods to add, delete and access elements at both sides of the underlying data sequence. As such, the [Deque](#) can also be used to mimic both stacks and queues. For the standard S3 interface, see [deque\(\)](#).

Details

This class inherits from class [Container\(\)](#) and extends it by `popleft` and `peek` methods, and `reverse` and `rotate` functionality.

Super classes

`container::Iterable` -> `container::Container` -> Deque

Methods

Public methods:

- [Deque\\$addleft\(\)](#)
- [Deque\\$peek\(\)](#)
- [Deque\\$peekleft\(\)](#)
- [Deque\\$popleft\(\)](#)
- [Deque\\$rev\(\)](#)
- [Deque\\$rotate\(\)](#)
- [Deque\\$clone\(\)](#)

Method `addleft()`: Add element to left side of the Deque.

Usage:

```
Deque$addleft(value, name = NULL)
```

Arguments:

`value` value of ANY type to be added to the Deque.
`name` character optional name attribute of the value.

Returns: the Deque object.

Method `peek()`: Peek at last element of the Deque.

Usage:

```
Deque$peek(default = NULL)
```

Arguments:

`default` returned default value if Deque is empty.

Returns: element 'peeked' on the right

Method `peekleft()`: Peek at first element of the Deque.

Usage:

`Deque$peekleft(default = NULL)`

Arguments:

`default` returned default value if Deque is empty.

Returns: element 'peeked' on the left

Method `popleft()`: Delete and return element from the left side of the [Deque\(\)](#).

Usage:

`Deque$popleft()`

Returns: element 'popped' from the left side of the [Deque\(\)](#)

Method `rev()`: Reverse all elements of the [Deque\(\)](#) in-place.

Usage:

`Deque$rev()`

Returns: the [Deque\(\)](#) object.

Method `rotate()`: Rotate all elements `n` steps to the right. If `n` is negative, rotate to the left.

Usage:

`Deque$rotate(n = 1L)`

Arguments:

`n` integer number of steps to rotate

Returns: returns the [Deque\(\)](#) object.

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

Usage:

`Deque$clone(deep = FALSE)`

Arguments:

`deep` Whether to make a deep clone.

See Also

[Container\(\)](#), [deque\(\)](#)

Examples

```

d = Deque$new(1, 2, s = "a", v = 1:3)
d$addleft(0)
d$peekleft()
d$peek()

d$popleft()
d$rev()

d$rotate()
d$rotate(2)
d$rotate(-3)

```

DequeS3

*Deque - Double-Ended Queue***Description**

Dequeues are a generalization of stacks and queues typically with methods to add, remove and access elements at both sides of the underlying data sequence. As such, the [deque](#) can also be used to mimic both stacks and queues.

Usage

```

deque(...)

as.deque(x)

is.deque(x)

```

Arguments

<code>...</code>	initial elements put into the Deque.
<code>x</code>	R object of ANY type for as.deque() and is.deque() or of class Deque for the S3 methods.

Details

Methods that alter [Deque](#) objects usually come in two versions providing either copy or reference semantics where the latter start with 'ref_' to note the reference semantic, for example, `add()` and `ref_add()`.

- `deque(...)` initializes and returns an object of class Deque
- `as.deque(x)` coerces `x` to a deque.
- `is.deque(x)` returns TRUE if `x` is of class Deque and FALSE otherwise.

- $x + y$ combines x and y into a new deque by appending y to x .
- $x - y$ element-wise removes all items of y from x , given the element was contained in x .
- `addleft(.x, ...)` adds (possibly named) elements to left side of $.x$.
- `ref_addleft(.x, ...)` same as `addleft(.x, ...)` but adds by reference.
- `peek(x, default = NULL)` peek at last element. If x is empty, return `default`.
- `peekleft(x, default = NULL)` peek at first element. If x is empty, return `default`.
- `ref_pop(.x)` pop last element. If $.x$ is empty, an error is given.
- `ref_popleft(.x)` pop first element. If $.x$ is empty, an error is given.
- `rev(x)` and `ref_rev(x)` reverses all elements being done on a copy or in place, respectively.
- `rotate(x, n)` rotate all elements n steps to the right, If n is negative, rotate to the left.

See Also

See [container\(\)](#) for all inherited methods. For the full class documentation see [Deque\(\)](#) and it's superclass [Container\(\)](#).

Examples

```
d = deque(1, 2, s = "a", v = 1:3)
is.deque(d)
print(d)
length(d)
names(d)
as.list(d)
rev(d)

l = list(0, 1)
d2 = as.deque(l)
d + d2
c(d, d2) # same as d + d2
d2 + d
d - d2
c(d2, d) # same as d2 + d
d2 - d
# Math
d = deque(1, 2, -(3:5))
d
abs(d)
cumsum(d)
round(d)
exp(d)

# Summary
range(d)
min(d)
```

```

max(d)

d1 = deque(1, 1:2)
d2 = deque(2, 1:2)
d1 + d2      # same as c(d1, d2)
d2 + d1      # same as c(d2, d1)

d1 - d2
d2 - d1
d1 - d1

d = deque(0)
add(d, a = 1, b = 2)      # |0, a = 1, b = 2|
addleft(d, a = 1, b = 2)  # |b = 2, a = 1, 0|

d = deque(1, 2, 3)
peek(d)
peekleft(d)
peek(deque())
peek(deque(), default = 0)
peekleft(deque(), default = 0)
d = deque(1, 2, 3)
ref_pop(d)
print(d)
ref_popleft(d)
print(d)

try({
  ref_pop(deque()) # pop at empty Deque
})

d = deque(a = 1, b = 2, 3)
rev(d)
print(d)
ref_rev(d)
print(d)

d = deque(1, 2, 3, 4)
rotate(d)
rotate(d, n = 2)

```

Dict

Dict Class

Description

The `Dict()` resembles Python's dict type, and is implemented as a specialized associative `Container()`. For the standard S3 interface, see `dict()`.

Details

This class inherits from class `Container()` and overrides some methods to account for the associative key-value pair semantic. Internally, all key-value pairs are stored in a hash-table and the elements are always sorted lexicographically by their keys.

Super classes

```
container::Iterable -> container::Container -> Dict
```

Methods

Public methods:

- `Dict$new()`
- `Dict$add()`
- `Dict$discard_at()`
- `Dict$get()`
- `Dict$keys()`
- `Dict$remove()`
- `Dict$replace()`
- `Dict$set()`
- `Dict$sort()`
- `Dict$update()`
- `Dict$values()`
- `Dict$clone()`

Method `new()`: Dict constructor

Usage:

```
Dict$new(...)
```

Arguments:

... initial elements put into the Dict

Returns: returns the Dict

Method `add()`: If name not yet in Dict, insert value at name, otherwise signal an error.

Usage:

```
Dict$add(name, value)
```

Arguments:

name character variable name under which to store value.

value the value to be added to the Dict.

Returns: the Dict object

Method `discard_at()`: Discard value at given index. If index is not found, the operation is ignored.

Usage:

Dict\$discard_at(index)

Arguments:

index character or numeric index

Returns: the Dict object

Method get(): This function is deprecated. Use [at2\(\)](#) instead.

Usage:

Dict\$get(key)

Arguments:

key character name of key.

Returns: If key in Dict, return value at key, else throw error.

Method keys(): Get all keys.

Usage:

Dict\$keys()

Returns: character vector of all keys.

Method remove(): This function is deprecated. Use [delete\(\)](#) instead.

Usage:

Dict\$remove(key)

Arguments:

key character name of key.

Returns: If key in Dict, remove it, otherwise raise an error.

Method replace(): Replace one element by another element. Search for occurrence of old and, if found, replace it by new. If old does not exist, an error is signaled.

Usage:

Dict\$replace(old, new)

Arguments:

old element to be replaced

new element to be put instead of old

Returns: the Dict object

Method set(): This function is deprecated. Use [replace\(\)](#) instead.

Usage:

Dict\$set(key, value, add = FALSE)

Arguments:

key character name of key.

value the value to be set

add logical if TRUE the value is set regardless whether key already exists in Dict.

Returns: returns the Dict

Method `sort()`: Sort elements according to their keys. This function is deprecated as keys are now always sorted.

Usage:

```
Dict$sort(decr = FALSE)
```

Arguments:

`decr` logical if TRUE sort in decreasing order.

Returns: returns the Dict

Method `update()`: Add elements of other to this if the name is not in the Dict and update elements with existing names.

Usage:

```
Dict$update(other)
```

Arguments:

`other` Iterable object used to update this.

Returns: returns the updated Dict object.

Method `values()`: Get Container values

Usage:

```
Dict$values()
```

Returns: a copy of all elements in a list

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

Usage:

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

Arguments:

`deep` Whether to make a deep clone.

See Also

[Container\(\)](#), [dict\(\)](#)

Examples

```
d = Dict$new(o = "one", na = NA, a = 1)
d
d$keys()

d$add("li", list(1, 2))
d$discard_at("na")
d$replace(1, 9)

d2 = Dict$new(a = 0, b = 1)
d$update(d2)
```

dict.table	<i>Combining Dict and data.table</i>
------------	--------------------------------------

Description

The `dict.table` is a combination of `dict` and `data.table` and basically can be considered a `data.table` with unique column names and an extended set of functions to add, extract and remove data columns with the goal to further facilitate code development using `data.table`. A `dict.table` object provides all `dict` and `data.table` functions and operators at the same time.

Usage

```
dict.table(...)

as.dict.table(x, ...)

## S3 method for class 'data.table'
as.dict.table(x, copy = TRUE, ...)

is.dict.table(x)

## S3 method for class 'dict.table'
rbind(x, ...)

## S3 method for class 'dict.table'
cbind(x, ...)
```

Arguments

<code>...</code>	elements put into the <code>dict.table</code> and/or additional arguments to be passed on.
<code>x</code>	any R object or a <code>dict.table</code> object.
<code>copy</code>	if TRUE creates a copy of the <code>data.table</code> object otherwise works on the passed object by reference.

Details

Methods that alter `dict.table` objects usually come in two versions providing either copy or reference semantics where the latter start with 'ref_' to note the reference semantic, for example, `add()` and `ref_add()`.

- `dict.table(...)` initializes and returns a `dict` object.
- `as.dict.table(x, ...)` coerce `x` to a `dict.table`
- `is.dict.table(x)` check if `x` is a `dict.table`
- `add(.x, ...)` and `ref_add(.x, ...)` add columns to `.x`. If the column name already exists, an error is given.

- `at(.x, ...)` returns the columns at the given indices. Indices can be letters or numbers or both. All columns must exist.
- `at2(x, index)` returns the column at the given index or signals an error if not found.
- `clear(x)` and `ref_clear(x)` remove all elements from `x`.
- `clone(x)` create a copy of `x`.
- `delete_at(.x, ...)` and `ref_delete_at(.x, ...)` find and remove columns either by name or index (or both). If one or more columns don't exist, an error is signaled.
- `discard_at(.x, ...)` and `ref_discard_at(.x, ...)` find and remove columns either by name or index (or both). Invalid column indices are ignored.
- `has(x, column)` check if some column is in `dict.table` object.
- `has_name(x, name)` check if `x` has the given column name.
- `is_empty(x)` TRUE if object is empty otherwise FALSE
- `peek_at(x, ..., .default = NULL)` returns the columns at the given indices or (if not found) columns with the given default value.
- `peek_at2(x, index, default = NULL)` return column named `index` if it exist otherwise the given default value. If the default length does not match the number of rows, it is recycled accordingly and a warning is given, unless the default value has a length of 1, in which case recycling is done silently.
- `ref_pop(.x, index)` return element at given column index and remove the column from the `dict.table` object.
- `rename(.x, old, new)` and `ref_rename(.x, old, new)` rename one or more columns from old to new, respectively, by copy and in place (i.e. by reference).
- `replace_at(.x, ..., .add = FALSE)` and `ref_replace_at(.x, ..., .add = FALSE)` replace values at given indices. If a given index is invalid, an error is signaled unless `.add` was set to TRUE.
- `update(object, other)` and `ref_update(object, other)` adds columns of other dict that are not yet in object and replaces the values at existing columns.

See Also

[dict](#), [data.table](#)

Examples

```
# Some basic examples using some typical data.table and dict operations.
# The constructor can take the 'key' argument known from data.table:
require(data.table)
dit = dict.table(x = rep(c("b","a","c"), each = 3), y = c(1,3,6), key = "y")
print(dit)
setkey(dit, "x") # sort by 'x'
print(dit)
(add(dit, "v" = 1:9)) # add column v = 1:9
dit[y > 5]
(ref_discard_at(dit, "x")) # discard column 'x'

try(at(dit, "x")) # index 'x' not found
try(replace_at(dit, x = 0)) # cannot be replaced, if it does not exist

dit = replace_at(dit, x = 0, .add = TRUE) # ok - re-adds column 'x' with all 0s
peek_at(dit, "x") # glance at column 'x'
has_name(dit, "x") # TRUE
ref_pop(dit, "x") # get column and remove it
has_name(dit, "x") # FALSE

# Copy and reference semantics when coercing *from* a data.table
dat = data.table(a = 1, b = 2)
dit = as.dict.table(dat)
is.dict.table(dit) # TRUE
is.dict.table(dat) # FALSE
ref_replace_at(dit, "a", 9)
dit[["a"]] # 9
dat[["a"]] # 1
dit.dat = as.dict.table(dat, copy = FALSE) # init by reference
ref_replace_at(dit.dat, "a", 9)
dit.dat[["a"]] # 9
is.dict.table(dit.dat) # TRUE
is.dict.table(dat) # TRUE now as well!

# Coerce from dict
d = dict(a = 1, b = 1:3)
as.dict.table(d)

dit = dict.table(a = 1:2, b = 1:2)
rbind(dit, dit)

# rbind ...
dit = dict.table(a = 1:2, b = 1:2)
rbind(dit, dit)

# ... can be mixed with data.tables
dat = data.table(a = 3:4, b = 3:4)
rbind(dit, dat) # yields a dict.table
rbind(dat, dit) # yields a data.table
```

```

# cbind ...
dit = dict.table(a = 1:2, b = 1:2)
dit2 = dict.table(c = 3:4, d = 5:6)
cbind(dit, dit2)

# ... can be mixed with data.tables
dat = data.table(x = 3:4, y = 3:4)
cbind(dit, dat)

dit = dict.table(a = 1:3)
add(dit, b = 3:1, d = 4:6)

try(add(dit, a = 7:9)) # column 'a' already exists

dit = dict.table(a = 1:3, b = 4:6)
at(dit, "a")
at(dit, 2)
at(dit, "a", 2)
try(at(dit, "x"))      # index 'x' not found
try(at(dit, 1:3))      # index 3 exceeds length of dict.table

dit = dict.table(a = 1:3, b = 4:6)
at2(dit, 1)
at2(dit, "a")
at2(dit, 2)
try(at2(dit, "x"))     # index 'x' not found
try(at2(dit, 5))       # index 5 exceeds length of dict.table

dit = dict.table(a = 1, b = 2)
clear(dit)
dit
ref_clear(dit)
dit

d = dict.table(a = 1:2, b = 3:4)
d2 = clone(d)
ref_clear(d)
print(d2)

(dit = as.dict.table(head(sleep)))
delete_at(dit, "ID")
delete_at(dit, "ID", 1)

try({
  delete_at(dit, "foo") # Column 'foo' not in dict.table
})

dit = as.dict.table(head(sleep))
discard_at(dit, "ID")
discard_at(dit, "ID", 1)
discard_at(dit, "foo") # ignored

dit = dict.table(a = 1:3, b = as.list(4:6))

```

```

has(dit, 1:3)          # TRUE
has(dit, 4:6)          # FALSE
has(dit, as.list(4:6)) # TRUE

dit = dict.table(a = 1, b = 2)
has_name(dit, "a")     # TRUE
has_name(dit, "x")     # FALSE

d = dict.table(a = 1:4, b = 4:1)
is_empty(d)
is_empty(clear(d))

dit = dict.table(a = 1:3, b = 4:6)
peek_at(dit, "a")
peek_at(dit, 1)
peek_at(dit, 3)
peek_at(dit, "x")
peek_at(dit, "x", .default = 0)
peek_at(dit, "a", "x", .default = 0)

dit = dict.table(a = 1:3, b = 4:6)
peek_at2(dit, "a")
peek_at2(dit, 1)
peek_at2(dit, 3)
peek_at2(dit, 3, default = 9)
peek_at2(dit, "x")
peek_at2(dit, "x", default = 0)

dit = dict.table(a = 1:3, b = 4:6)
ref_pop(dit, "a")
ref_pop(dit, 1)

try({
  ref_pop(dit, "x") # index 'x' not found
})

dit = dict.table(a = 1, b = 2, c = 3)
rename(dit, c("a", "b"), c("a1", "y"))
print(dit)
ref_rename(dit, c("a", "b"), c("a1", "y"))
print(dit)

dit = dict.table(a = 1:3)
replace_at(dit, "a", 3:1)

try({
  replace_at(dit, "b", 4:6)          # column 'b' not in dict.table
})
replace_at(dit, "b", 4:6, .add = TRUE) # ok, adds column

# Update parts of tables (second overwrites columns of the first)
dit1 = dict.table(a = 1:2, b = 3:4)
dit2 = dict.table(      b = 5:6, c = 8:9)

```



```
update(dit1, dit2)
update(dit2, dit1)
```

DictS3

A Dictionary

Description

The [Dict](#) initially was developed to resemble Python's dict type, but by now offers both more features and flexibility, for example, by providing both associative key-value pair as well as positional array semantics. It is implemented as a specialized associative [Container](#) thus sharing all [Container](#) methods with some of them being adapted to account for the key-value pair semantic. All elements must be named.

Usage

```
dict(...)

as.dict(x)

is.dict(x)
```

Arguments

...	elements put into the Dict.
x	R object of ANY type for as.dict() and is.dict() or of class Dict for the S3 methods.

Details

Internally, all key-value pairs are stored in a hash-table and the elements are sorted lexicographically by their keys. Methods that alter Dict objects usually come in two versions providing either copy or reference semantics where the latter start with 'ref_' to note the reference semantic, for example, [add\(\)](#) and [ref_add\(\)](#).

- [dict\(...\)](#) initializes and returns an object of class Dict
- [as.dict\(x\)](#) coerces x to a dictionary
- [is.dict\(x\)](#) returns TRUE if x is of class Dict and FALSE otherwise.
- $x + y$ combines x and y into a new dict by updating x by y (see also [\[update\(\)\]](#)).
- $x - y$ removes all keys from x that appear in y.
- $x \& y$ returns a copy of x keeping only the keys that are common in both (key intersection), that is, all keys in x that do not exist in y are removed.

- `x | y` returns a copy of `x` extended by all elements of `y` that are stored at keys (or names) that do not exist in `x`, thereby combining the keys of both objects (set union of keys).
- `add(.x, ...)` and `ref_add(.x, ...)` adds `key = value` pairs to `.x`. If any of the keys already exists, an error is given.
- `replace(.x, old, new)` and `ref_replace(.x, old)` try to find element `old` and replace it with element `new`. If `old` does not exist, an error is raised.
- `update(object, other)` and `ref_update(object, other)` adds elements of `other` dict for keys not yet in `object` and replaces the values of existing keys.

See Also

See `container()` for all inherited methods. For the full class documentation see [Dict](#) and its superclass [Container](#).

Examples

```
d = dict(b = "one", a = 1, f = mean, na = NA)
print(d)
names(d)

try(dict(a = 1, 2))  # all elements must be named

# Coercion
as.dict(list(A = 1:3, B = "b"))
as.dict(c(x = 1, y = "x", z = 2 + 3))
# Math
d = dict(a = rnorm(1), b = rnorm(1))
abs(d)
cumsum(d)
round(d)
exp(d)

# Summary
range(d)
min(d)
max(d)

d1 = dict(a = 1, b = list(1, 2))
d2 = dict(a = 2, b = list(1, 2))
d1 + d2      # same as update(d, d2)
d2 + d1      # same as update(d2, d)
try({
  c(d1, d2)  # duplicated keys are not allowed for Dict
})
d1 - d2
d2 - d1
d1 - d1

d1 = dict(a = 1, b = 2)
```

```

d2 = dict(a = 10, x = 4)
d1 & d2      # {a = 1}

d1 | d2      # {a = 1, b = 2, x = 4}

d = dict(a = 1)
add(d, b = 2, co = container(1:3))

try(add(d, a = 7:9)) # key 'a' already in Dict

d = dict(a = 1, b = "z")
replace(d, 1, 1:5)
replace(d, "z", "a")

try({
  replace(d, "a", 2)          # old element ("a") is not in Dict
})

d1 = dict(a = 1, b = 2)
d2 = dict(b = 0, c = 3)
update(d1, d2) # {a = 1, b = 0, c = 3}
update(d2, d1) # {a = 1, b = 2, c = 3}

```

discard

Discard Container Elements

Description

Search and remove an element from an object. If the element is not found, ignore the attempt.

Usage

```

discard(.x, ...)

ref_discard(.x, ...)

## S3 method for class 'Container'
discard(.x, ...)

## S3 method for class 'Container'
ref_discard(.x, ...)

```

Arguments

```

.x          any R object.
...         elements to be discarded.

```

Value

For Container, an object of class Container (or one of the respective derived classes).

Examples

```
s = setnew("a", num = 1:3, data = iris)
print(s)
discard(s, 1:3, "a")
discard(s, iris)
discard(s, "b") # ignored
```

discard_at

Discard Elements at Indices

Description

Search and remove values at given indices, which can be numeric or character or both. Invalid indices are ignored.

Usage

```
discard_at(.x, ...)

ref_discard_at(.x, ...)

## S3 method for class 'Container'
discard_at(.x, ...)

## S3 method for class 'Container'
ref_discard_at(.x, ...)

## S3 method for class 'dict.table'
discard_at(.x, ...)

## S3 method for class 'dict.table'
ref_discard_at(.x, ...)
```

Arguments

.x any R object.
 ... indices at which values are to be discarded.

Value

For Container, an object of class Container (or one of the respective derived classes).

For dict.table, an object of class dict.table.

Examples

```
co = container(a = 1, b = 2, 3)
discard_at(co, "a", "b")      # [3]
discard_at(co, 1:2)           # [3]
discard_at(co, "a", 3)        # [b = 2]
discard_at(co, "x")           # ignored

dit = as.dict.table(head(sleep))
discard_at(dit, "ID")
discard_at(dit, "ID", 1)
discard_at(dit, "foo") # ignored
```

has	<i>Check for Element</i>
-----	--------------------------

Description

Check for Element

Usage

```
has(x, ...)

## S3 method for class 'Container'
has(x, elem, ...)

## S3 method for class 'dict.table'
has(x, column, ...)
```

Arguments

x	any R object.
...	additional arguments to be passed to or from methods.
elem	some element to be found.
column	vector of values with the same length as the number of rows of the <code>dict.table</code> .

Value

TRUE if element is in x and otherwise FALSE.
 For `dict.table`, TRUE if column exists in x otherwise FALSE.

See Also

[has_name\(\)](#)

Examples

```
co = container(1, 2, mean)
has(co, 1)           # TRUE
has(co, mean)        # TRUE
has(co, 1:2)         # FALSE

dit = dict.table(a = 1:3, b = as.list(4:6))
has(dit, 1:3)        # TRUE
has(dit, 4:6)        # FALSE
has(dit, as.list(4:6)) # TRUE
```

has_name	<i>Check for Name</i>
----------	-----------------------

Description

Check for Name

Usage

```
has_name(x, name)

## S3 method for class 'Container'
has_name(x, name)

## S3 method for class 'dict.table'
has_name(x, name)
```

Arguments

- x any R object.
- name character the name to be found.

Value

TRUE if name is in x and otherwise FALSE.
For dict.table TRUE if the dict.table objects has the given column name, otherwise FALSE.

See Also

[has\(\)](#)

Examples

```
co = container(a = 1, 2, f = mean)
has_name(co, "a")    # TRUE
has_name(co, "f")    # TRUE
has_name(co, "2")    # FALSE

dit = dict.table(a = 1:2, b = 3:4)
has_name(dit, "a")   # TRUE
has_name(dit, "x")   # FALSE
```

is_empty	<i>Check if Object is Empty</i>
----------	---------------------------------

Description

Check if Object is Empty

Usage

```
is_empty(x)

## S3 method for class 'Container'
is_empty(x)

## S3 method for class 'dict.table'
is_empty(x)
```

Arguments

x any R object.

Value

TRUE if object is empty otherwise FALSE.

Examples

```
co = container(1, 2)
is_empty(co)
is_empty(clear(co))

d = dict.table(a = 1:4, b = 4:1)
is_empty(d)
is_empty(clear(d))
```

Iterable*Iterable abstract class interface*

Description

An [Iterable](#) is an object that provides an [iter\(\)](#) method, which is expected to return an [Iterator](#) object. This class defines the abstract class interface such that each class inheriting this class provides an [iter\(\)](#) method and must implement a private method `create_iter()`, which must return an [Iterator](#) object.

Methods

Public methods:

- [Iterable\\$new\(\)](#)
- [Iterable\\$iter\(\)](#)
- [Iterable\\$clone\(\)](#)

Method `new()`: `Iterable` is an abstract class and thus cannot be instantiated.

Usage:

```
Iterable$new()
```

Method `iter()`: Create iterator

Usage:

```
Iterable$iter()
```

Returns: returns the `Iterator` object.

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

Usage:

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

Arguments:

`deep` Whether to make a deep clone.

Author(s)

Roman Pahl

See Also

[Iterator](#) and [Container](#)

Iterator

Iterator Class

Description

An Iterator is an object that allows to iterate over sequences. It implements `next_iter` and `get_value` to iterate and retrieve the value of the sequence it is associated with. For the standard S3 interface, see `iter()`.

Methods

Public methods:

- `Iterator$new()`
- `Iterator$begin()`
- `Iterator$get_value()`
- `Iterator$get_next()`
- `Iterator$has_next()`
- `Iterator$has_value()`
- `Iterator$length()`
- `Iterator$pos()`
- `Iterator$next_iter()`
- `Iterator$print()`
- `Iterator$reset_iter()`
- `Iterator$clone()`

Method `new()`: Iterator constructor

Usage:

```
Iterator$new(x, .subset = .subset2)
```

Arguments:

`x` object to iterate over

`.subset` accessor function

Returns: invisibly returns the Iterator object

Method `begin()`: set iterator to the first element of the underlying sequence unless length of sequence is zero, in which case it will point to nothing.

Usage:

```
Iterator$begin()
```

Returns: invisibly returns the Iterator object

Method `get_value()`: get value where the iterator points to

Usage:

```
Iterator$get_value()
```

Returns: returns the value the Iterator is pointing at.

Method `get_next()`: get next value

Usage:

`Iterator$get_next()`

Returns: increments the iterator and returns the value the Iterator is pointing to.

Method `has_next()`: check if iterator has more elements

Usage:

`Iterator$has_next()`

Returns: TRUE if iterator has next element else FALSE

Method `has_value()`: check if iterator points at value

Usage:

`Iterator$has_value()`

Returns: TRUE if iterator points at value otherwise FALSE

Method `length()`: iterator length

Usage:

`Iterator$length()`

Returns: number of elements to iterate

Method `pos()`: get iterator position

Usage:

`Iterator$pos()`

Returns: integer if iterator has next element else FALSE

Method `next_iter()`: increment iterator

Usage:

`Iterator$next_iter()`

Returns: invisibly returns the Iterator object

Method `print()`: print method

Usage:

`Iterator$print()`

Method `reset_iter()`: reset iterator to '0'

Usage:

`Iterator$reset_iter()`

Returns: invisibly returns the Iterator object

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

Usage:

`Iterator$clone(deep = FALSE)`

Arguments:

`deep` Whether to make a deep clone.

Author(s)

Roman Pahl

Examples

```
# Numeric Vector
v = 1:3
it = Iterator$new(v)
it

try(it$get_value()) # iterator does not point at a value

it$has_value()
it$has_next()
it$next_iter()
it$get_value()
it$get_next()
it$get_next()
it
it$has_next()
it$begin()
it$get_value()
it$reset_iter()

# Works by reference for Container
co = Container$new(1, 2, 3)
it = co$iter()
it$get_next()
co$discard(2)
it
it$get_value()
co$discard(1)
it
it$get_value()
it$begin()
```

iterS3*Iterate over Sequences*

Description

An Iterator is an object that allows to iterate over sequences. It implements `next_iter()` and `get_value()` to iterate and retrieve the value of the sequence it is associated with. For documentation of the methods see [Iterator](#).

Usage

```
iter(x, ...)
```

```
## S3 method for class 'Container'
iter(x, ...)

## Default S3 method:
iter(x, ...)

is.iterator(x)

is.iterable(x)

begin(it)

get_value(it)

get_next(it)

has_next(it)

has_value(it)

pos(it)

next_iter(it)

reset_iter(it)

## S3 method for class 'Iterator'
length(x)
```

Arguments

x	an object of class Iterable or any other R object. In the latter case, x will always be coerced to a base R list prior to creating the Iterator .
...	other parameters passed to or from methods
it	Iterator object

Value

length returns the number of elements that can be iterated over.

See Also

For the class documentation see [Iterator](#).

Examples

```
# Numeric Vector
v = 1:3
it = iter(v)
```

```
it

try(it$get_value()) # iterator does not point at a value

has_value(it)
has_next(it)
next_iter(it)
get_value(it)
get_next(it)
get_next(it)
it
has_next(it)
begin(it)
get_value(it)
reset_iter(it)

# Works on copy of Container
co = container(1, 2, 3)
it = iter(co)
get_next(it)
ref_discard(co, 2)
co
it
get_next(it)
ref_clear(co)
co
it
get_next(it)
begin(it)
```

OpsArithmetic

Arithmetic Operators

Description

Binary arithmetic operators for [Container\(\)](#) objects and derived classes.

Usage

```
## S3 method for class 'Container'
x + y

## S3 method for class 'Container'
x - y

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

## S3 method for class 'Deque'
```

```

x - y

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

## S3 method for class 'Dict'
x - y

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

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

```

Arguments

`x, y` Depending on the operator at least one must be of class `Container()` or the respective derived class and the other at least be coercible to the respective class.

Value

For `Container`, `x + y` combines `x` and `y` into a new container by appending `y` to `x`.

For `Container`, `x - y` element-wise discards all items of `y` from `x`, given the element was contained in `x`. The result is always a container.

For `Deque`, `x + y` combines `x` and `y` into a new deque by appending `y` to `x`.

For `Deque`, `x - y` element-wise removes all items of `y` from `x`, given the element was contained in `x`.

For `Dict`, `x + y` combines `x` and `y` into a new dict by updating `x` by `y` (see also `[update()]`).

For `Dict`, `x - y` removes all keys from `x` that appear in `y`.

For `Set`, `x + y` performs the set union.

For `Set`, `x - y` performs the set difference.

Examples

```

c1 = container(1, 1:2)
c2 = container(2, 1:2)
c1 + c2      # same as c(c1, c2)
c2 + c1      # same as c(c2, c1)

c1 - c2
c2 - c1
c1 - c1
# Arithmetic
d1 = deque(1, 1:2)
d2 = deque(2, 1:2)
d1 + d2      # same as c(d1, d2)
d2 + d1      # same as c(d2, d1)

d1 - d2

```

```

d2 - d1
d1 - d1

# Arithmetic
d1 = dict(a = 1, b = list(1, 2))
d2 = dict(a = 2, b = list(1, 2))
d1 + d2      # same as update(d, d2)
d2 + d1      # same as update(d2, d)
try({
c(d1, d2)    # duplicated keys are not allowed for Dict
})
d1 - d2
d2 - d1
d1 - d1

# Arithmetic
s1 = setnew(1, 1:2)
s2 = setnew(2, 1:2)
s1 + s2      # same as s1 | s2 or c(c1, s2)
s2 + s1      # same

s1 - s2
s2 - s1

```

OpsCompare

*Comparison Operators***Description**

Binary comparison operators for [Container\(\)](#) objects and derived classes.

Usage

```

## S3 method for class 'Container'
x == y

## S3 method for class 'Container'
x != y

## S3 method for class 'Container'
x < y

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

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

```

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

Arguments

`x, y` at least one must be a `Container()` object (or an object of one of the derived classes) while the other must be at least iterable.

Details

- `x == y` is TRUE if the contents of `x` and `y` are lexicographically *equal*.
- `x != y` is TRUE if the contents of `x` and `y` are *not equal*.
- `x < y` is TRUE if the contents of `x` are lexicographically *less* than the contents of `y`.
- `x <= y` is TRUE if the contents of `x` are lexicographically *less* than or *equal* to the contents of `y`.

Examples

```
c1 = container(1, 2, 3)
c2 = container(1, 3, 2)
c1 == c1      # TRUE
c1 != c2      # TRUE
c1 <= c1      # TRUE
c1 == c2      # FALSE
c1 < c2       # TRUE
c1 < container(2) # TRUE
c1 < container() # FALSE
```

OpsExtract

Extract Parts of a Container Object

Description

Extract parts of a Container object similar to R's base extract operators on lists.

Usage

```
## S3 method for class 'Container'
x[...]

## S3 method for class 'Container'
x[[i]]
```

Arguments

`x` Container object from which to extract elements.

`i, ...` indices specifying elements to extract. Indices are numeric or character vectors or a list containing both.

Details

[selects multiple values. The indices can be numeric or character or both. They can be passed as a vector or list or, for convenience, just as a comma-separated sequence (see Examples). Non-existing indices are ignored.

[[selects a single value using a numeric or character index.

Examples

```
co = container(a = 1, b = 2, c = 3, d = 4)
co[1:2]
co[1, 4]
co["d", 2]
co[list("d", 2)]
co[0:10]

co = container(a = 1, b = 2)
co[[1]]
co[["a"]]
co[["x"]]
```

OpsLogic

Logic Operators

Description

Binary logic operators for `Container()` objects and derived classes.

Usage

```
## S3 method for class 'Dict'
x & y

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

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

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

Arguments

x, y	Depending on the operator at least one must be of class <code>Container()</code> or the respective derived class and the other at least be coercible to the respective class.
------	---

Examples

```
d1 = dict(a = 1, b = 2)
d2 = dict(a = 10, x = 4)
d1 & d2      # {a = 1}
```

OpsReplace

*Replace Parts of a Container***Description**

Replace parts of a Container object similar to R's base replace operators on lists.

Usage

```
## S3 replacement method for class 'Container'
x[i] <- value

## S3 replacement method for class 'Container'
x[[i]] <- value

## S3 replacement method for class 'Container'
x$name <- value
```

Arguments

x	Container object in which to replace elements.
i	indices specifying elements to replace. Indices are numeric or character vectors or a list containing both.
value	the replacing value of ANY type
name	character string (possibly backtick quoted)

Details

[<- replaces multiple values. The indices can be numeric or character or both. They can be passed as a vector or list. Values can be added by 'replacing' at new indices, which only works for character indices.

[[<- replaces a single value at a given numeric or character index. Instead of an index, it is also possible to replace certain elements by passing the element in curly braces (see Examples), that is, the object is searched for the element and then the element is replaced by the value.

\$<- replaces a single element at a given name.

Examples

```

co = container(a = 1, b = "bar")
(co[1:2] <- 1:2)

try({
  co[3] <- 3 # index out of range
})
(co[list(1, "b")] <- 3:4) # mixed numeric/character index

co = container(a = 1, b = 2)
co[[1]] <- 9
co[["b"]] <- 8
co[["x"]] <- 7
co$z <- 99
print(co)

# Replace 8 by 0
co[[{8}]] <- 0
print(co)

co = container(a = 1, b = "bar")
co$f <- 3
co$b <- 2
co

```

OrderedSet

*OrderedSet Class***Description**

The [OrderedSet](#) is a [Set](#) where all elements are always ordered.

Details

The order of elements is determined sequentially as follows:

- element's length
- whether it is an atomic element
- the element's class(es)
- by numeric value (if applicable)
- it's representation when printed
- the name of the element in the [Set](#)

Super classes

[container::Iterable](#) -> [container::Container](#) -> [container::Set](#) -> OrderedSet

Methods

Public methods:

- [OrderedSet\\$new\(\)](#)
- [OrderedSet\\$add\(\)](#)
- [OrderedSet\\$clone\(\)](#)

Method `new()`: OrderedSet constructor

Usage:

```
OrderedSet$new(...)
```

Arguments:

... initial elements put into the OrderedSet

Returns: returns the OrderedSet object

Method `add()`: Add element

Usage:

```
OrderedSet$add(value, name = NULL)
```

Arguments:

value value of ANY type to be added to the OrderedSet.

name character optional name attribute of the value.

Returns: the OrderedSet object.

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

Usage:

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

Arguments:

deep Whether to make a deep clone.

See Also

[Container](#), [Set](#)

Examples

```
s1 = OrderedSet$new(2, 1)
s1
```

peek	<i>Peek at Left or Right of a Deque</i>
------	---

Description

Try to access first or last element and return some default value if not found. In contrast to `[at2()]`, this function provides a less stricter element access, that is, it remains valid even if peeked elements don't exist.

Usage

```
peekleft(x, default = NULL)

peek(x, default = NULL)

## S3 method for class 'Deque'
peek(x, default = NULL)

## S3 method for class 'Deque'
peekleft(x, default = NULL)
```

Arguments

x	a Deque object.
default	value to be returned if peeked value does not exist.

Details

peek peek at last element of a Deque.
peekleft peek at first element of a Deque.

Value

The first (peekleft) or last (peek) element.

See Also

[at2\(\)](#) for strict element extraction

Examples

```
# Deque
d = deque(1, 2, 3)
peek(d)
peekleft(d)
peek(deque())
peek(deque(), default = 0)
peekleft(deque(), default = 0)
```

peek_at

*Peek at Indices***Description**

Try to access elements and return default values if not found. In contrast to `[at()]`, this function provides a less stricter element access, that is, it remains valid even if elements don't exist.

Usage

```
peek_at(.x, ...)

## S3 method for class 'Container'
peek_at(.x, ..., .default = NULL)

## S3 method for class 'dict.table'
peek_at(.x, ..., .default = NULL)
```

Arguments

<code>.x</code>	an R object of the respective class.
<code>...</code>	indices of elements to be extracted
<code>.default</code>	value to be returned if peeked value does not exist.

Details

`peek_at` tries to access specific values.

Value

For `Container`, returns the value at the given indices or (if not found) the given default value.

For `dict.table`, returns the columns at the given indices or (if not found) columns with the given default value.

See Also

[at\(\)](#) for strict element extraction

Examples

```
# Container
co = container(a = 1, 2, b = 3, 4)
peek_at(co, 1)
peek_at(co, "a")
peek_at(co, "x")
peek_at(co, "x", .default = 0)
peek_at(co, "a", "x", 2, 9, .default = -1)
```

```
# Dict
d = dict(a = 1, b = 1:3)
peek_at(d, "b")
peek_at(d, "x")
peek_at(d, "x", .default = 4:7)

# dict.table
dit = dict.table(a = 1:3, b = 4:6)
peek_at(dit, "a")
peek_at(dit, 1)
peek_at(dit, 3)
peek_at(dit, "x")
peek_at(dit, "x", .default = 0)
peek_at(dit, "a", "x", .default = 0)
```

peek_at2

Peek at Single Index

Description

Try to access element and return some default value if not found. In contrast to `[at2()]`, this function provides a less stricter element access, that is, it remains valid even if peeked elements don't exist.

Usage

```
peek_at2(x, index, default = NULL)

## S3 method for class 'Container'
peek_at2(x, index, default = NULL)

## S3 method for class 'dict.table'
peek_at2(x, index, default = NULL)
```

Arguments

<code>x</code>	an R object of the respective class.
<code>index</code>	character name or numeric position of the sought value.
<code>default</code>	value to be returned if peeked value does not exist.

Value

For `Container`, returns the value at the given index or (if not found) the given default value.

For `dict.table`, returns the column named `index` if it exist otherwise the given default value. If the default length does not match the number of rows, it is recycled accordingly and a warning is given, unless the default value has a length of 1, in which case recycling is done silently.

See Also

[at2\(\)](#) for strict element extraction

Examples

```
# Container
co = container(a = 1, 2, b = 3, 4)
peek_at2(co, 1)
peek_at2(co, "a")
peek_at2(co, "x")
peek_at2(co, "x", default = 0)

# Dict
d = dict(a = 1, b = 1:3)
peek_at2(d, "b")
peek_at2(d, "x")
peek_at2(d, "x", default = 4:7)

# dict.table
dit = dict.table(a = 1:3, b = 4:6)
peek_at2(dit, "a")
peek_at2(dit, 1)
peek_at2(dit, 3)
peek_at2(dit, 3, default = 9)
peek_at2(dit, "x")
peek_at2(dit, "x", default = 0)
```

 pop

Get and Remove Element

Description

Search and return an element and remove it afterwards from the object. If the element is not found, signal an error.

Usage

```
ref_pop(.x, ...)
```

```
ref_popleft(.x, ...)
```

```
## S3 method for class 'Deque'
ref_pop(.x, ...)
```

```
## S3 method for class 'Deque'
ref_popleft(.x, ...)
```

```
## S3 method for class 'Container'
```



```
ref_pop(.x, index, ...)

## S3 method for class 'dict.table'
ref_pop(.x, index, ...)
```

Arguments

<code>.x</code>	any R object.
<code>...</code>	additional arguments to be passed to or from methods.
<code>index</code>	character name or numeric position of value to be popped

Details

All functions work by reference, that is, the original object is altered. `ref_pop(.x)` tries to access specific values.

`ref_popleft(.x)` pops first element of a Deque.

Value

For Deque the first (`ref_popleft`) or last (`ref_pop`) element of the deque after it was removed.

For Container the value at the given index after it was removed from the Container object. If index is not found, an error is raised.

For `dict.table`, returns the column at the given index after it was removed from the `dict.table`. If column does not exist, an error is raised.

See Also

[peek\(\)](#)

Examples

```
# Deque
d = deque(1, 2, 3)
ref_pop(d)
ref_popleft(d)

try({
  ref_pop(deque()) # pop at empty Deque
})

# Container
co = container(a = 1, b = 1:3, d = "foo")
ref_pop(co, "b")
ref_pop(co, 1)

try({
  ref_pop(co, "x") # index 'x' not found
})

# dict.table
```

```

dit = dict.table(a = 1:3, b = 4:6)
ref_pop(dit, "a")
ref_pop(dit, 1)

try({
  ref_pop(dit, "x") # index 'x' not found
})

```

rename	<i>Rename Elements Safely</i>
--------	-------------------------------

Description

Search for old name and replace it by new name. If either the old name does not exist or the name would result in a name-clash with an already existing name, an error is signaled.

Usage

```

rename(.x, old, new)

ref_rename(.x, old, new)

## S3 method for class 'Container'
rename(.x, old, new)

## S3 method for class 'dict.table'
rename(.x, old, new)

## S3 method for class 'dict.table'
ref_rename(.x, old, new)

## Default S3 method:
rename(.x, old, new)

```

Arguments

.x	dict.table object
old	character old name
new	character new name

Details

The passed old and new names can be vectors but always must have the same length and must be unique to prevent double-renaming.

rename uses copy semantics while ref_rename works by reference, that is, it renames elements in place.

Value

For standard R vectors renames old to new and returns the renamed vector.

For Container, an object of class Container (or one of the respective derived classes).

For dict.table renames key old to new in place (i.e. by reference) and invisibly returns the `dict.table()` object.

Examples

```
# Container
co = container(a = 1, b = 2, 3)
rename(co, c("a", "b"), c("a1", "y"))
print(co)
ref_rename(co, c("a", "b"), c("a1", "y"))
print(co)

# dict.table
dit = dict.table(a = 1, b = 2, c = 3)
rename(dit, c("a", "b"), c("a1", "y"))
print(dit)
ref_rename(dit, c("a", "b"), c("a1", "y"))
print(dit)
```

replace

Replace Values in Containers Safely

Description

Try to find and replace elements and signal an error if not found, unless it is stated to explicitly add the element (see option add).

Usage

```
replace(.x, ...)
```

```
ref_replace(.x, ...)
```

```
## S3 method for class 'Container'
replace(.x, old, new, add = FALSE, ...)
```

```
## S3 method for class 'Container'
ref_replace(.x, old, new, add = FALSE, ...)
```

```
## S3 method for class 'Dict'
replace(.x, old, new, ...)
```

```
## S3 method for class 'Dict'
ref_replace(.x, old, new, ...)
```

Arguments

.x	any R object.
...	additional arguments to be passed to or from methods.
old	old element to be found and replaced.
new	the new element replacing the old one.
add	logical if FALSE (default) and element was not found, an error is given. In contrast, if set to TRUE the new element is added regardless of whether it is used as a replacement for an existing element or just added as a new element.

Details

replace uses copy semantics while ref_replace works by reference.

Value

For Container, an object of class Container (or one of the respective derived classes).
For Dict an object of class Dict.

Examples

```
co = container("x", 9)
replace(co, 9, 0)
replace(co, "x", 0)
try({
  replace(co, "z", 0)           # old element ("z") is not in Container
})
replace(co, "z", 0, add = TRUE) # just add the zero without replacement

d = dict(a = 1, b = "z")
replace(d, 1, 1:5)
replace(d, "z", "a")

try({
  replace(d, "a", 2)           # old element ("a") is not in Dict
})
```

replace_at	<i>Replace Values at Indices Safely</i>
------------	---

Description

Try to find and replace elements at given indices and signal an error if not found, unless it is stated to explicitly add the element (see option add).

Usage

```

replace_at(.x, ...)

ref_replace_at(.x, ...)

## S3 method for class 'Container'
replace_at(.x, ..., .add = FALSE)

## S3 method for class 'Container'
ref_replace_at(.x, ..., .add = FALSE)

## S3 method for class 'dict.table'
replace_at(.x, ..., .add = FALSE)

## S3 method for class 'dict.table'
ref_replace_at(.x, ..., .add = FALSE)

```

Arguments

<code>.x</code>	any R object.
<code>...</code>	either name = value pairs or two vectors/lists with names/values to be replaced.
<code>.add</code>	logical if FALSE (default) and index is invalid, an error is given. If set to TRUE the new element is added at the given index regardless whether the index existed or not. Indices can consist of numbers or names or both, except when adding values at new indices, which is only allowed for names.

Details

`replace_at` uses copy semantics while `ref_replace_at` works by reference.

Value

For `Container`, an object of class `Container` (or one of the respective derived classes).

For `dict.table` an object of class `dict.table`.

Examples

```

co = container(a = 0, b = "z")
replace_at(co, a = 1, b = 2)
replace_at(co, 1:2, 1:2)           # same
replace_at(co, c("a", "b"), list(1, 2)) # same

try({
  replace_at(co, x = 1)             # names(s) not found: 'x'
})
replace_at(co, x = 1, .add = TRUE)  # ok (adds x = 1)

dit = dict.table(a = 1:3, b = 4:6)

```

```
replace_at(dit, a = 3:1)
replace_at(dit, 1, 3:1)           # same
replace_at(dit, "a", 3:1)        # same
replace_at(dit, a = 3:1, b = 6:4)
replace_at(dit, 1:2, list(3:1, 6:4)) # same

try({
  replace_at(dit, x = 1)           # column(s) not found: 'x'
})
replace_at(dit, x = 1, .add = TRUE) # ok (adds column)
```

rev	<i>Reverse Elements</i>
-----	-------------------------

Description

rev provides a reversed version of its argument.

Usage

```
ref_rev(x)

## S3 method for class 'Deque'
ref_rev(x)

## S3 method for class 'Deque'
rev(x)
```

Arguments

x Deque object

Details

rev uses copy semantics while ref_rev works by reference, that is, it reverse all elements in place.

Value

For Deque, an object of class Deque

See Also

```
base::rev\(\)
```

Examples

```
d = deque(a = 1, b = 2, 3)
rev(d)
print(d)
ref_rev(d)
print(d)
```

rotate

*Rotate Elements***Description**

Rotate all elements *n* steps to the right. If *n* is negative, rotate to the left.

Usage

```
rotate(x, n = 1L)

ref_rotate(x, n = 1L)

## S3 method for class 'Deque'
rotate(x, n = 1L)

## S3 method for class 'Deque'
ref_rotate(x, n = 1L)
```

Arguments

x any R object.

n integer number of steps to rotate

Details

While `rotate` uses copy semantics, `ref_rotate` works by reference, that is, rotates in place on the original object.

Value

For `Deque` returns the rotated `Deque()` object.

Examples

```
d = deque(1, 2, 3, 4)
rotate(d)
rotate(d, n = 2)
```

Set

*Set Class***Description**

The [Set](#) is considered and implemented as a specialized [Container](#), that is, elements are always unique in the [Container](#) and it provides typical set operations such as union and intersect. For the standard S3 interface, see [setnew\(\)](#).

Super classes

[container::Iterable](#) -> [container::Container](#) -> Set

Methods**Public methods:**

- [Set\\$new\(\)](#)
- [Set\\$add\(\)](#)
- [Set\\$diff\(\)](#)
- [Set\\$intersect\(\)](#)
- [Set\\$union\(\)](#)
- [Set\\$is_equal\(\)](#)
- [Set\\$is_subset\(\)](#)
- [Set\\$is_proper_subset\(\)](#)
- [Set\\$values\(\)](#)
- [Set\\$clone\(\)](#)

Method [new\(\)](#): Set constructor

Usage:

[Set\\$new\(...\)](#)

Arguments:

... initial elements put into the Set

Returns: returns the Set object

Method [add\(\)](#): Add element

Usage:

[Set\\$add\(value, name = NULL\)](#)

Arguments:

value value of ANY type to be added to the Set.

name character optional name attribute of the value.

Returns: the Set object.

Method [diff\(\)](#): Set difference

Usage:

Set\$diff(s)

Arguments:

s Set object to 'subtract'

Returns: the Set object updated as a result of the set difference between this and s.

Method intersect(): Set intersection

Usage:

Set\$intersect(s)

Arguments:

s Set object to 'intersect'

Returns: the Set object as a result of the intersection of this and s.

Method union(): Set union

Usage:

Set\$union(s)

Arguments:

s Set object to be 'unified'

Returns: the Set object as a result of the union of this and s.

Method is_equal(): Set equality

Usage:

Set\$is_equal(s)

Arguments:

s Set object to compare against

Returns: TRUE if this is equal to s, otherwise FALSE

Method is_subset(): Set proper subset

Usage:

Set\$is_subset(s)

Arguments:

s Set object to compare against

Returns: TRUE if this is subset of s, otherwise FALSE

Method is_proper_subset(): Set subset

Usage:

Set\$is_proper_subset(s)

Arguments:

s Set object to compare against

Returns: TRUE if this is proper subset of s, otherwise FALSE

Method `values()`: Get Set values

Usage:

`Set$values()`

Returns: elements of the set as a base list

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

[Container](#), [setnew\(\)](#)

Examples

```
s1 = Set$new(1, 2)
s1
s1$add(1)
s1$add(3)
s2 = Set$new(3, 4, 5)
s1$union(s2)
s1

s1 = Set$new(1, 2, 3)
s1$intersect(s2)
s1

s1$diff(s2)
s1$diff(s1)
s1
```

SetS3

Set and ordered Set

Description

The [Set](#) is considered and implemented as a specialized [Container](#), that is, Set elements are always unique. It provides typical set operations such as union and intersect.

Usage

```
setnew(..., .ordered = FALSE)
```

```
as.set(x)
```

```
as.orderedset(x)
```

```
is.set(x)
```

```
is.orderedset(x)
```

Arguments

<code>...</code>	initial elements put into the Set.
<code>.ordered</code>	logical if TRUE all elements in the Set will be ordered.
<code>x</code>	R object of ANY type for as.set() and is.set() or of class Set for the S3 methods.

Details

Methods that alter [Set](#) objects usually come in two versions providing either copy or reference semantics where the latter start with 'ref_' to note the reference semantic, for example, [add\(\)](#) and [ref_add\(\)](#).

- `setnew(...)` initializes and returns a [Set\(\)](#) object.
- `as.set(x)` coerces x to a set.
- `as.orderedset(x)` coerces x to an ordered set.
- `is.set(x)` returns TRUE if x is of class Set and FALSE otherwise.
- `is.orderedset(x)` returns TRUE if x is of class OrderedSet and FALSE otherwise.
- `x & y` performs the set intersection of x and y
- `x | y` performs the set union of x and y

See Also

See [container\(\)](#) for all inherited methods. For the full class documentation see [Set](#) and it's super-class [Container](#).

Examples

```

s = setnew(1, b = NA, 1:3, c = container("a", 1))
is.set(s)
print(s)
length(s)
names(s)
as.list(s)
unpack(s)  # flatten recursively similar to unlist

so = setnew(2, 1, .ordered = TRUE)
print(so)
add(so, 0)
# Math
s = setnew(5:3, 1, 2)
s
abs(s)
cumsum(s)
round(s)
exp(s)

# Summary
range(s)
min(s)
max(s)

s1 = setnew(1, 1:2)
s2 = setnew(2, 1:2)
s1 + s2      # same as s1 | s2 or c(c1, s2)
s2 + s1      # same

s1 - s2
s2 - s1

s1 = setnew(1, b = 2)
s2 = setnew(1, b = 4)
s1 & s2      # {1}

s1 | s2      # {1, b = 2, b = 4}

```

unpack

Unpack Nested Objects

Description

Similar to `unlist()` recursively unpacks any (possibly nested) structure into a flat list. In contrast to `unlist()`, `unpack()` also works with (possibly nested) `Container()` objects. In principle, it works for any object that can be transformed to a list via `as.list`.

Usage

```
unpack(x, recursive = TRUE, use.names = TRUE)
```

Arguments

x	any R object
recursive	logical descend recursively into nested objects?
use.names	logical Should names be preserved?

Value

a list

update	<i>Update Object with Elements from Another Object</i>
--------	--

Description

Takes an object and updates it with values from another object by replacing the values at existing names and adding values at new names of the other object. A common use case is to update parameter lists.

Usage

```
ref_update(object, other, ...)

## S3 method for class 'Container'
update(object, other, ...)

## S3 method for class 'Container'
ref_update(object, other, ...)

## S3 method for class 'dict.table'
update(object, other, ...)

## S3 method for class 'dict.table'
ref_update(object, other, ...)

## S3 method for class 'list'
update(object, other, ...)
```

Arguments

object	any R object
other	any object of the same type as object
...	additional arguments to be passed to or from methods.

Details

update uses copy semantics while `ref_update` works by reference, that is, updates in place.

Value

For `Container`, an object of class `Container` (or one of the respective derived classes).

For `dict.table` an object of class `dict.table`.

For `list`, an updated object of class `list`.

Examples

```
d1 = dict(a = 1, b = 2)
d2 = dict(      b = 0, c = 3)
update(d1, d2) # {a = 1, b = 0, c = 3}
update(d2, d1) # {a = 1, b = 2, c = 3}

dit1 = dict.table(a = 1:2, b = 3:4)
dit2 = dict.table(      b = 5:6, c = 8:9)
update(d1, d2)
update(d2, d1)

l1 = list(1, b = 2)
l2 = list(  b = 0, c = 3)
update(l1, l2)
update(l2, l1)
```

Index

!=.Container (OpsCompare), 55
+.Container (OpsArithmetic), 53
+.Deque (OpsArithmetic), 53
+.Dict (OpsArithmetic), 53
+.Set (OpsArithmetic), 53
-.Container (OpsArithmetic), 53
-.Deque (OpsArithmetic), 53
-.Dict (OpsArithmetic), 53
-.Set (OpsArithmetic), 53
<.Container (OpsCompare), 55
<=.Container (OpsCompare), 55
==.Container (OpsCompare), 55
>.Container (OpsCompare), 55
>=.Container (OpsCompare), 55
[.Container (OpsExtract), 56
[<-.Container (OpsReplace), 58
[[.Container (OpsExtract), 56
[[<-.Container (OpsReplace), 58
\$<-.Container (OpsReplace), 58
&.Dict (OpsLogic), 57
&.Set (OpsLogic), 57

add, 3, 3
add(), 17, 36, 75
addleft, 4, 5
as.cont (Containers3), 16
as.container, 17
as.container (Containers3), 16
as.deque (DequeS3), 30
as.deque(), 30
as.dict (DictS3), 41
as.dict(), 41
as.dict.table (dict.table), 36
as.list(), 28
as.list.Container (Containers3), 16
as.orderedset (SetS3), 74
as.set (SetS3), 74
as.set(), 75
at, 5
at(), 62

at2, 6
at2(), 34, 61, 64

base::rev(), 70
begin(iterS3), 51

cbind.dict.table (dict.table), 36
clear, 7
clone, 8
cont (Containers3), 16
Container, 3, 8, 9, 16, 17, 19, 41, 42, 48, 60, 72, 74, 75
container (Containers3), 16
Container(), 11, 28, 29, 31–33, 35, 53–57, 76
container(), 9, 15, 31, 42, 75
container::Container, 28, 33, 59, 72
container::Iterable, 9, 28, 33, 59, 72
container::Set, 59
container_options, 22
Containers3, 16
count, 24

delete, 25
delete(), 14, 28, 34
delete_at, 25
deprecated, 27
Deque, 3, 4, 9, 15, 16, 28, 28, 30
deque, 19, 30
deque (DequeS3), 30
Deque(), 29, 31, 71
deque(), 28, 29
DequeS3, 30
Dict, 3, 9, 15, 16, 27, 32, 41, 42
dict, 19, 36, 37
dict (DictS3), 41
Dict(), 32
dict(), 32, 35
dict.table, 3, 8, 36, 36
dict.table(), 67
DictS3, 41

- discard, [43](#)
- discard_at, [44](#)
- empty (deprecated), [27](#)
- empty(), [27](#)
- format(), [14](#)
- get_next(iterS3), [51](#)
- get_value(iterS3), [51](#)
- get_value(), [51](#)
- getContainerOption(container_options), [22](#)
- has, [45](#)
- has(), [46](#)
- has_name, [46](#)
- has_name(), [45](#)
- has_next(iterS3), [51](#)
- has_value(iterS3), [51](#)
- is.container, [17](#)
- is.container(ContainerS3), [16](#)
- is.deque(DequeS3), [30](#)
- is.deque(), [30](#)
- is.dict(DictS3), [41](#)
- is.dict(), [41](#)
- is.dict.table(dict.table), [36](#)
- is.iterable(iterS3), [51](#)
- is.iterator(iterS3), [51](#)
- is.orderedset(SetS3), [74](#)
- is.set(SetS3), [74](#)
- is.set(), [75](#)
- is_empty, [47](#)
- is_empty(), [12](#), [27](#)
- iter(iterS3), [51](#)
- iter(), [48](#), [49](#)
- Iterable, [9](#), [15](#), [48](#), [48](#), [52](#)
- Iterator, [48](#), [49](#), [51](#), [52](#)
- iterS3, [51](#)
- keys (deprecated), [27](#)
- length(), [14](#), [27](#)
- length.Container(ContainerS3), [16](#)
- length.Iterator(iterS3), [51](#)
- list, [16](#), [17](#), [52](#)
- names.Container(ContainerS3), [16](#)
- names<- .Container(ContainerS3), [16](#)
- next_iter(iterS3), [51](#)
- next_iter(), [51](#)
- OpsArithmetic, [53](#)
- OpsCompare, [55](#)
- OpsExtract, [56](#)
- OpsLogic, [57](#)
- OpsReplace, [58](#)
- OrderedSet, [59](#), [59](#)
- peek, [61](#)
- peek(), [65](#)
- peek_at, [62](#)
- peek_at(), [5](#)
- peek_at2, [63](#)
- peek_at2(), [7](#)
- peekleft(peek), [61](#)
- pop, [64](#)
- pos(iterS3), [51](#)
- rbind.dict.table(dict.table), [36](#)
- ref_add, [3](#)
- ref_add(add), [3](#)
- ref_add(), [17](#), [36](#), [75](#)
- ref_addleft, [5](#)
- ref_addleft(addleft), [4](#)
- ref_clear(clear), [7](#)
- ref_delete(delete), [25](#)
- ref_delete_at(delete_at), [25](#)
- ref_discard(discard), [43](#)
- ref_discard_at(discard_at), [44](#)
- ref_pop(pop), [64](#)
- ref_popleft(pop), [64](#)
- ref_rename(rename), [66](#)
- ref_replace(replace), [67](#)
- ref_replace_at(replace_at), [68](#)
- ref_rev(rev), [70](#)
- ref_rotate(rotate), [71](#)
- ref_update(update), [77](#)
- remove(), [28](#)
- rename, [66](#)
- replace, [67](#)
- replace(), [34](#)
- replace_at, [68](#)
- reset_iter(iterS3), [51](#)
- rev, [70](#)
- rotate, [71](#)
- Set, [3](#), [9](#), [15](#), [16](#), [59](#), [60](#), [72](#), [72](#), [74](#), [75](#)

Set(), [75](#)
setnew, [19](#)
setnew (SetS3), [74](#)
setnew(), [27](#), [72](#), [74](#)
SetS3, [74](#)
size (deprecated), [27](#)
size(), [27](#)
sortkey (deprecated), [27](#)
sortkey(), [27](#)

type(), [28](#)

unlist(), [76](#)
unpack, [76](#)
unpack(), [76](#)
update, [77](#)

values (deprecated), [27](#)
values(), [28](#)