Package ‘RCX’

May 30, 2024

Type Package
Title R package implementing the Cytoscape Exchange (CX) format
Version 1.8.0
Description Create, handle, validate, visualize and convert networks in the Cytoscape exchange (CX) format to standard data types and objects.
The package also provides conversion to and from objects of igraph and graphNEL. The CX format is also used by the NDEx platform, an online commons for biological networks, and the network visualization software Cytoscape.
License MIT + file LICENSE
Depends R (>= 4.2.0)
Imports jsonlite, plyr, igraph, methods
RoxygenNote 7.1.2
BiocViews Pathways, DataImport, Network
Suggests BiocStyle, testthat, knitr, rmarkdown, base64enc, graph
VignetteBuilder knitr
Encoding UTF-8
URL https://github.com/frankkramer-lab/RCX
BugReports https://github.com/frankkramer-lab/RCX/issues
git_url https://git.bioconductor.org/packages/RCX
git_branch RELEASE_3_19
git_last_commit d46f370
git_last_commit_date 2024-04-30
Repository Bioconductor 3.19
Date/Publication 2024-05-29
Author Florian Auer [aut, cre] (<https://orcid.org/0000-0002-5320-8900>)
Maintainer Florian Auer <florian.auer@informatik.uni-augsburg.de>
Contents

.addAspectNameToJson . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 4
.addAttributeData . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 4
.aspectClass . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 5
.convertRawList . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 6
.createAttributeAspect . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 7
.createCyVpPorD . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 8
.errorCodes . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 9
.filterBy . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 11
.format . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 11
.infer . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 13
.json2RDataType . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 14
.jsonL . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 15
.jsonV . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 16
.log . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 16
.mergeAttributesAspect . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 17
.mergeIdAspect . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 18
.modClass . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 19
.pasteC . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 20
.renameDF . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 21
.stop . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 22
.summaryAspect . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 24
.transformListLength<- . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 25
.transformVLD . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 25
.validateAttributesAspect . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 26
.validateCyVisualPropertyPandD . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 27
.aspectClasses . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 27
.CartesianLayout . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 29
.checks . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 30
.convert-data-types-and-values . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 34
.Convert-Names-and-Classes . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 35
.convert2json . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 36
.countElements . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 37
.custom-print . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 38
.CyGroups . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 40
.CyHiddenAttributes . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 41
.CyNetworkRelations . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 43
.CySubNetworks . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 45
.CyTableColumn . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 46
.CyVisualProperties . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 47
.CyVisualProperty . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 50
.CyVisualPropertyDependencies . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 52
.CyVisualPropertyMappings . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 54
.CyVisualPropertyProperties . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 56
dot_test . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 58
.EdgeAttributes . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 62
.Edges . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 64
<table>
<thead>
<tr>
<th>Function</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>getCyVisualProperty</td>
<td>65</td>
</tr>
<tr>
<td>graphNEL</td>
<td>68</td>
</tr>
<tr>
<td>hasIds</td>
<td>70</td>
</tr>
<tr>
<td>idProperty</td>
<td>71</td>
</tr>
<tr>
<td>Igraph</td>
<td>72</td>
</tr>
<tr>
<td>jsonToRCX</td>
<td>74</td>
</tr>
<tr>
<td>markAttributeColumn</td>
<td>76</td>
</tr>
<tr>
<td>markRefColumn</td>
<td>77</td>
</tr>
<tr>
<td>maxId</td>
<td>77</td>
</tr>
<tr>
<td>Meta-data</td>
<td>78</td>
</tr>
<tr>
<td>NetworkAttributes</td>
<td>80</td>
</tr>
<tr>
<td>NodeAttributes</td>
<td>82</td>
</tr>
<tr>
<td>Nodes</td>
<td>85</td>
</tr>
<tr>
<td>RCX</td>
<td>86</td>
</tr>
<tr>
<td>RCX-object</td>
<td>87</td>
</tr>
<tr>
<td>rcxToJson</td>
<td>90</td>
</tr>
<tr>
<td>readCX</td>
<td>93</td>
</tr>
<tr>
<td>referredBy</td>
<td>95</td>
</tr>
<tr>
<td>refersTo</td>
<td>96</td>
</tr>
<tr>
<td>setExtension</td>
<td>97</td>
</tr>
<tr>
<td>summary</td>
<td>98</td>
</tr>
<tr>
<td>toCX</td>
<td>100</td>
</tr>
<tr>
<td>updateCartesianLayout</td>
<td>101</td>
</tr>
<tr>
<td>updateCyGroups</td>
<td>104</td>
</tr>
<tr>
<td>updateCyHiddenAttributes</td>
<td>106</td>
</tr>
<tr>
<td>updateCyNetworkRelations</td>
<td>109</td>
</tr>
<tr>
<td>updateCySubNetworks</td>
<td>112</td>
</tr>
<tr>
<td>updateCyTableColumn</td>
<td>115</td>
</tr>
<tr>
<td>updateCyVisualProperties</td>
<td>118</td>
</tr>
<tr>
<td>updateCyVisualProperty</td>
<td>122</td>
</tr>
<tr>
<td>updateEdgeAttributes</td>
<td>125</td>
</tr>
<tr>
<td>updateEdges</td>
<td>128</td>
</tr>
<tr>
<td>updateMetaDataProperties</td>
<td>130</td>
</tr>
<tr>
<td>updateNetworkAttributes</td>
<td>131</td>
</tr>
<tr>
<td>updateNodeAttributes</td>
<td>134</td>
</tr>
<tr>
<td>updateNodes</td>
<td>137</td>
</tr>
<tr>
<td>validate</td>
<td>138</td>
</tr>
<tr>
<td>visualize</td>
<td>141</td>
</tr>
<tr>
<td>writeCX</td>
<td>142</td>
</tr>
<tr>
<td>writeHTML</td>
<td>143</td>
</tr>
</tbody>
</table>
.addAspectNameToJson  Add the aspect name to the JSON

Description
Add the aspect name to the JSON

Usage
.addAspectNameToJson(json, name)

Arguments
json character; preformated json
name character; name of the aspect

Value
character; character of json object

Note
Internal function only for convenience

Examples
json = '{bla:"BLA", blubb:"BLUBB"}
RCX:::.addAspectNameToJson(json, "foo")

.addAttributeData  Add attribute data to an igraph object

Description
Not only simply add the name-value pairs, but also:
• unlist lists if indicated by isList column
• renames name="name" to "attribute$name"
• puts subnetwork id at the and of the attribute name
• adds a data type as attribute$dataType if not string, boolean or double

Usage
.addAttributeData(ig, attributeRef, attribute)
.aspectClass

Arguments

- `ig` igraph object
- `attributeRef` reference name; "node", "edge" or "network"
- `attribute` an attribute aspect

Value

igraph object

Note

Internal function only for convenience

Examples

NULL

Description

Get the class of aspects

Usage

.aspectClass(x)

Arguments

- `x` a potential aspect

Value

The aspect class name or NA if it’s not an aspect

Note

Internal function only for convenience
Examples

```r
x = list(a="foo", b="bar")
class(x)
# [1] "list"
## Not run:
.addClass(x) <- .CLS$nodes
class(x)
# [1] "NodesAspect" "list"

.aspectClass(x)
# [1] "NodesAspect"

.removeClass(x) <- "NodesAspect"

.aspectClass(x)
# [1] "NA"

## End(Not run)
```

---

**.convertRawList**  
Convert a list of vectors to a character vector with pasted elements

### Description

Convert a list of vectors to a character vector with pasted elements

### Usage

```
.convertRawList(l, keepNa = TRUE)
```

### Arguments

- **l**: unnamed list
- **keepNa**: logical; whether to keep NA values or replace it with an empty list

### Value

character

### Note

Internal function only for convenience

### Examples

```r
l = list(NA, c(2,3), 5)
RCX:::.convertRawList(l)
```
Create a default *AttributeAspect

Description

Some aspects like NodeAttributesAspect or EdgeAttributesAspect use a key-value scheme. This function helps in constructing while avoiding repetition.

Usage

```r
createAttributeAspect(
  propertyOf,  # integer; refers to the IDs of an other aspect
  name,        # character; key of the attribute
  value,       # character; value of the attribute
  dataType,    # character (optional);
  isList,      # logical (optional);
  subnetworkId = NULL,  # integer (optional); CySubNetworks
  log = ""
)
```

Arguments

- `propertyOf` integer; refers to the IDs of an other aspect
- `name` character; key of the attribute
- `value` character; value of the attribute
- `dataType` character (optional);
- `isList` logical (optional);
- `subnetworkId` integer (optional); CySubNetworks

Value

*AttributeAspect prototype object

Note

Internal function only for convenience

See Also

*mergeIdAspect, mergeAttributesAspect, CySubNetworks*

Examples

```r
NULL
```
Helper to create structure for classes CyVisualPropertyProperties and CyVisualPropertyDependencies

Arguments
name character, optional; name of the properties
value character or named character; value of the properties
.log character (optional); name of the calling function used in logging

Value
data.frame

Note
Internal function only for convenience

See Also
Used in CyVisualPropertyProperties, CyVisualPropertyDependencies and CyVisualPropertyMappings

Examples
## Not run:
data1 = c(NODE_BORDER_STROKE="SOLID", NODE_BORDER_WIDTH="1.5")
.createCyVpPorD(value=data1)

key1 = c("NODE_BORDER_STROKE", "NODE_BORDER_WIDTH")
value1 = c("SOLID", "1.5")
.createCyVpPorD(key1, value1)

# Result for either:
#   name value
# 1 NODE_BORDER_STROKE SOLID
# 2 NODE_BORDER_WIDTH 1.5

## End(Not run)
.errorCodes

---

**.errorCodes**  
*Error codes used in this package*

---

**Description**

This function returns the error message to a given (internal) error code. For some codes, additional information for the message is needed.

**Usage**

```
```

**Arguments**

- `code` character; Error code.
- `info` character; Additional information used in some error codes.

**Value**

Full text for a given error code.

**Details**

**List of error codes:**

- `e404`:
  
  THIS ERROR SHOULD NEVER HAPPEN!!!

- `ErrorCodeNotFound`:
  
  (!ERROR CODE NOT FOUND!!

- `graphNELedgesRequired`:
  
  RCX object requires edges to be converted to a graphNEL object!

- `idNonNeg`:
  
  Provided IDs (<info[1]>) must be non-negative!

- `idNotNum`:
  
  Provided IDs (<info[1]>) must be numeric!

- `idRefNotFound`:
  
  Provided IDs of <info[1]> don't exist in <info[2]>

- `idRefNotPresent`:
  
  <info[1]> not present as <info[2]>

- `igraphEdgesRequired`:  
  

RCX object requires edges to be converted to an igraph object!

**paramAllNull:**
At least one argument of `<info[1]>` must be set!

**paramDifferentLength:**
Arguments must have the same length!

**<info[1]>**

**paramListAllWrongClass:**
Not all elements of the list `<info[1]>` are of class "<info[2]>"!

**paramMissing:**
Missing arguments: `<info[1]>`

**paramMissingRCX:**
RCX object is missing!

**paramNa:**
Argument `<info[1]>` must not contain any NA values!

**paramNonNeg:**
All elements of `<info[1]>` must be non-negative!

**paramNotChar:**
All elements of `<info[1]>` must be characters!

**paramNotList:**
Argument `<info[1]>` must be a list!

**paramNotLog:**
All elements of `<info[1]>` must be logical!

**paramNotNamed:**
Object `<info[1]>` must have names!

**paramNotNum:**
All elements of `<info[1]>` must be numeric!

**paramNotUnique:**
Elements of `<info[1]>` must not contain duplicates!

**paramWrongValue:**
Argument `<info[1]>` only can take following values: `<info[2]>`

**validationFail:**
Aspect `<info[1]>` failed validation!
Check if the aspect is valid: validate(<info[1]>)

**wrongClass:**
Class of object `<info[1]>` is not "<info[2]>"!

**wrongClassOf:**
Class of object `<info[1]>` is not one of `<info[2]>`!

**Note**

Internal function only for convenience
.filterBy  

Filter several parameters for elements, that match to a given name in a given param

Description

Filter several parameters for elements, that match to a given name in a given param

Usage

.filterBy(name, param, ...)

Arguments

name character; matching value
param character; in which param in ...
...

Value

list with only matching elements of all parameters

Note

Internal function only for convenience

Examples

po=c("match","not","some","other","match")
prop=c(1,2,3,4,5)
derp=c("bla","blubb","bla","blubb","bla")
map=list("BLA","BLUBB","BLA","BLUBB","BLA")

RCX:::.filterBy("match", "po", po, prop, dep, map)

.format  

Format objects for error logging

Description

Format objects for error logging
Usage

- `.formatQuote(v)`
- `.formatComma(v)`
- `.formatParams(v, con = "and")`
- `.formatData(v)`
- `.formatLog(v, fname = c())`
- `.formatO(v, fname)`

Arguments

- `v` character vector; just some strings
- `fname` character; function name
- `.log` character; previous called functions

Value

character

Functions

- `.formatQuote`: add quotes to the vector elements: "<v[i]>
- `.formatO`: format a object name with its calling function

Note

Internal function only for convenience

Examples

```r
## Not run:
v <- c("one", "two", "three")
fname <- "foo"
.log <- c("foo1", "foo2", "foo3")

.formatQuote(v)
# [1] "\"one\"" "\"two\"" "\"three\"

.formatComma(v)
# [1] "one, two, three"
```
.infer

Infer the data type from values and check if the value elements are a list

Description

Each element has an R data type (i.e., class). If more than one element are present in one list element, it is marked as list.

Usage

.inferDataType(values)

.inferIsList(values)

Arguments

values vector or list of R data values

Details

.inferDataType infers the data type of the elements in the vector or list. .inferIsList infers for each element if it is a list. For a vector, the return therefore is FALSE for each element!

Value

character vector of data types or logical vector of list status

Functions

• .inferIsList: Infer, if the values are lists
Note

Internal function only for convenience

Examples

NULL

---

`.json2RDataType`  
*Get the data type and isList from JSON data*

Description

Get the data type and isList from JSON data

Usage

`.json2RDataType(dataType, default = "string")`

Arguments

dataType  
data type column from jsonToRCX => .jsonV

default  
default value for NA values (by default the values remain NA)

Value

`list(type=<character vector>, isList=<logical vector>)`

Note

Internal function only for convenience

Examples

```r
jsonD = c("boolean", "double", "integer", "long", "string",  
          "list_of_boolean", "list_of_double", "list_of_integer",  
          "list_of_long", "list_of_string")

RCX:::.json2RDataType(jsonD)
```
`.jsonL`  
_Return data as a list from a JSON list_

**Description**

Return data as a list from a JSON list

**Usage**

```r
.jsonL(
  data,
  acc,
  default = as.character(NA),
  unList = TRUE,
  returnAllDefault = TRUE
)
```

**Arguments**

- `data`: json list
- `acc`: accession name
- `default`: default return value
- `unList`: logical; whether to unlist the list elements (e.g. for a list of lists return a list of vectors)
- `returnAllDefault`: whether to return the vector if all values are the default value (or NULL instead)

**Value**

list

**Note**

Internal function only for convenience

**Examples**

```r
testData = list(list(n="CDKN1B"),
  list(n="ROCK1", r="BLA"),
  list(n="SHC1", r="BLUBB"),
  list(n="IRS1"))
RCX::.jsonL(testData, "r")
```
Description
Return data as a vector from a JSON list

Usage
.jsonV(data, acc, default = NA, returnAllDefault = TRUE)

Arguments
- data: json list
- acc: accession name
- default: default return value
- returnAllDefault: whether to return the vector if all values are the default value (or NULL instead)

Value
vector

Note
Internal function only for convenience

Examples

```r
testData = list(
  list(n="CDKN1B"),
  list(n="ROCK1", r="BLA"),
  list(n="SHC1", r="BLUBB"),
  list(n="IRS1"))
RCX:::.jsonV(testData, "r")
```

Description
Logging (printing) the results of test cases

Usage
.log(info, pass, sep = "...", spaceLine = FALSE)
Arguments

- **info**: character; description of the test case.
- **pass**: boolean; was the test passed?
- **sep**: character (default="..."); separates description from result.
- **spaceLine**: boolean (default=FALSE); should a blank line be added after.

Value

NULL, only prints the log

Note

Internal function only for convenience

Examples

```r
testPassed <- TRUE
testFailed <- FALSE
## Not run:
.log('testing something', testPassed)
# testing something...OK
.log('testing other stuff', testFailed, spaceLine=TRUE)
# testing other stuff...FAIL
#
.log('testing more', testPassed, " ", TRUE)
# testing more OK

## End(Not run)
```

Description

Some aspects like `NodeAttributesAspect` or `EdgeAttributesAspect` use a key-value scheme. This function helps in merging while avoiding repetition.

Usage

```r
.mergeAttributesAspect(
  firstAspect,
  secondAspect,
  replace = TRUE,
  stopOnDuplicates = FALSE,
  required = c("propertyOf", "name"),
  optional = "subnetworkId",
  .log = c()
)
```
### Arguments

- **firstAspect**: *AttributeAspect object; first aspect.
- **secondAspect**: *AttributeAspect object; second aspect.
- **replace**: logical (default: TRUE); should duplicate keys be replaced with values of the secondAspect
- **stopOnDuplicates**: logical (default: FALSE); whether to stop, if duplicate keys are found
- **required**: character (optional); names of required column names
- **optional**: character (optional); names of optional column names
- **.log**: character (optional); origin of the data used for error logging

### Value

*AttributeAspect object

### Note

Internal function only for convenience

### See Also

- `.mergeIdAspect`
- `.createAttributeAspect`

### Examples

NULL

---

### Description

Merges two aspects, that are both data.frames and of the same aspect class. If the idCol contains duplicates new ids (for secondAspect) are created (ids of firstAspect are kept), unless it is specified otherwise by saveOldIds.

### Usage

```r
.mergeIdAspect(
  firstAspect, secondAspect, idCol, uniqCols, stopOnDuplicates = FALSE, saveOldIds = TRUE, .log = c()
)
```
Arguments

firstAspect  data.frame; first aspect.
secondAspect data.frame; second aspect.
idCol  character; name of the column to merge on.
uniqCols  character; name of the column to be checked for uniqueness.
stopOnDuplicates  boolean (default=FALSE); whether to stop, if duplicates in uniqCols column are found.
saveOldIds  boolean (default=TRUE); whether to keep the IDs from secondAspect, if duplicates in uniqCols column are found.
.log  character (optional); name of the calling function used in logging

Value

data.frame of the merged aspects.

Note

The two aspects must be the same type of aspect (same aspect class)!
Internal function only for convenience

See Also

.mergeAttributesAspect, .createAttributeAspect

Examples

NULL

.modClass  Add or remove a class name from an object

Description

Add or remove a class name from an object

Usage

.addClass(x) <- value

.removeClass(x) <- value

Arguments

x  an R object.
value  character vector of length 1.
Value

an R object.

Note

Internal function only for convenience

Examples

```r
x = list(a="foo", b="bar")
class(x)
# [1] "list"
## Not run:
.addClass(x) <- "blubb"
class(x)
# [1] "blubb" "list"

.addClass(x) <- "blubb"
class(x)
# [1] "blubb" "list"

.removeClass(x) <- "blubb"
class(x)
# [1] "list"

## End(Not run)
```

Description

Concatenate as comma separated character vector

Usage

`.pasteC(x)`

Arguments

x character vector.

Value

character vector of length 1.

Note

Internal function only for convenience
Examples

```r
## Not run:
a <- c("one", "two", "three")
.pasteC(a)
# [1] "one, two, three"

## End(Not run)
```

### Description

Rename data.frame columns by key-value pairs in `rnames`.

### Usage

```
.renameDF(df, rnames)
```

### Arguments

- `df`: data.frame
- `rnames`: named character vector; `names(rnames)=colnames(df)`

### Value

df with new colnames; or NULL on error

### Note

Internal function only for convenience

### Examples

```
nodes = data.frame(id=c(0,1,2),
                  name=c("CDK1","CDK3"),
                  represents=c(NA,"bla",NA))
rnames = c(id="@id", name="n", represents="r")
RCX:::.renameDF(nodes, rnames)
```
Customized stop() function

Description

Customized stop() function

Usage

.stop(code, info = NULL, msg = NULL)

Arguments

code character; Error code.
info character; Additional information used in some error codes.
msg character;

Value

Does not have any return value, simply throws an error!

Details

List of error codes::

e404:
THIS ERROR SHOULD NEVER HAPPEN!!!

ErrorCodeNotFound:
##############################
## !!ERROR CODE NOT FOUND!! ##
##############################
requested error code:
<info[1]>

graphNELEdgesRequired:
RCX object requires edges to be converted to an graphNEL object!

idNonNeg:
Provided IDs (<info[1]> ) must be non-negative!

idNotNum:
Provided IDs (<info[1]> ) must be numeric!

idRefNotFound:
Provided IDs of <info[1]> don't exist in <info[2]>

idRefNotPresent:
<info[1]> not present as <info[2]>

igraphEdgesRequired:
RCX object requires edges to be converted to an igraph object!

**paramAllNull:**
At least one argument of <info[1]> must be set!

**paramDifferentLength:**
Arguments must have the same length!
  <info[1]>

**paramListAllWrongClass:**
Not all elements of the list <info[1]> are of class "<info[2]>"!

**paramMissing:**
Missing arguments: <info[1]>

**paramMissingRCX:**
RCX object is missing!

**paramNa:**
Argument <info[1]> must not contain any NA values!

**paramNonNeg:**
All elements of <info[1]> must be non-negative!

**paramNotChar:**
All elements of <info[1]> must be characters!

**paramNotList:**
Argument <info[1]> must be a list!

**paramNotLog:**
All elements of <info[1]> must be logical!

**paramNotNamed:**
Object <info[1]> must have names!

**paramNotNum:**
All elements of <info[1]> must be numeric!

**paramNotUnique:**
Elements of <info[1]> must not contain duplicates!

**paramWrongValue:**
Argument <info[1]> only can take following values: <info[2]>

**validationFail:**
Aspect (<info[1]>) failed validation!
Check if the aspect is valid: validate(<info[1]>)

**wrongClass:**
Class of object <info[1]> is not "<info[2]"!

**wrongClassOf:**
Class of object <info[1]> is not one of <info[2]>

---

**Note**

Internal function only for convenience
Examples

```r
## Not run:
.stop("paramMissingRCX")
# Error: .stop
# RCX object is missing!

.stop("paramNotUnique","idParamName")
# Error: .stop
# Provided IDs (idParamName) contain duplicates!

.stop("wrongClass",c("nodesAspect", "NodesAspect"))
# Error: .stop
# Class of object "nodesAspect" is not "NodesAspect"!

## End(Not run)
```

---

**.summaryAspect**

*Helper function to mark columns that are ids or reference ids*

**Description**

Helper function to mark columns that are ids or reference ids

**Usage**

```r
.summaryAspect(aspect)
```

**Arguments**

- `aspect` an aspect (data.frame)

**Value**

the aspect (data.frame)

**Note**

Internal function only for convenience

**Examples**

```r
dges = createEdges(source=c(1,1), target=c(2,3))
dges = RCX::.summaryAspect(edges)
class(edges$id)
```
.transformListLength<-

Transform an aspect with a list as column

Description
Transforms an aspect with a column, that is a list to force a custom format in summary generation. Only show the number of elements in the list elements.

Usage
.transformListLength(aspect) <- value

Arguments
aspect an aspect (data.frame)
value character; property

Value
the aspect (data.frame)

Note
Internal function only for convenience

Examples
df = data.frame(bla=c("a","b","c"))
df$blubb=list(c("a","b","c"),
   c(1,2),
   c(TRUE,FALSE,FALSE,TRUE,TRUE))
RCX:::.transformListLength(df) = "blubb"

summary(df)

.transformVLD

Transform an aspect with data type

Description
Transforms an aspect with value, dataType and isList columns to force a custom format in summary generation.
Usage

.transformVLD(aspect)

Arguments

aspect an aspect (data.frame)

Value

the aspect (data.frame)

Note

Internal function only for convenience

Examples

df = data.frame(bla=c("a","b","c"),
value=list("a",2,TRUE),
dataType=c("string","integer","boolean"),
isList=c(FALSE,FALSE,FALSE))
df = RCX:::.transformVLD(df)

summary(df)

.validateAttributesAspect

Helper for validating node- and edge-attributes aspects

Description

Helper for validating node- and edge-attributes aspects

Usage

.validateAttributesAspect(aspect, verbose = TRUE)

Arguments

aspect an RCX aspect
verbose logical; whether to print the test results.

Value

logical; whether the test passed

Note

Internal function only for convenience
Cytoscape visual property: List of property and dependency

Description
For both properties the checks are the same.

Usage
.validateCyVisualPropertyPandD(aspect, property, verbose = TRUE)
.validateListOfCyVisualPropertyPandD(aspect, property, verbose = TRUE)

Arguments
aspect either CyVisualPropertyProperties or CyVisualPropertyDependencies object
property character; name of the property
verbose logical; whether to print the test results.

Value
logical; whether the object passed all tests.

Functions
• .validateListOfCyVisualPropertyPandD: List of property and dependency objects

Note
Internal function only for convenience

aspectClasses aspectClasses and subAspectClasses

Description
To get the aspect classes it is advised to always use the getAspectClasses() function to ensure the correct functionality. aspectClasses and subAspectClasses contain the default RCX accession name and the classes of the corresponding (sub)aspect. The getAspectClasses() function standardizes access to the accession names and classes, and also allows to include installed extensions of the RCX data model. Only installed and loaded extensions are included in the result. New extensions should register on load using the setExtension function to be added to options()$RCX.options$extensions, and therefore to getAspectClasses().
Usage

aspectClasses
getAspectClasses(extensions = TRUE)

subAspectClasses
updateAspectClasses(aspectClasses = aspectClasses)

Arguments

extensions logical; whether to include aspect classes from extensions
aspectClasses named character; accession names and aspect classes

Format

An object of class character of length 14.
An object of class character of length 4.

Details

updateAspectClasses sets the default aspect classes in options(RCX.options), either from aspectClasses or manually provided options.

Value

named character; accession names and aspect classes

See Also

setExtension

Examples

## default aspect classes
aspectClasses

## get set aspect classes from options()
aspectClasses = getAspectClasses()

## get aspect classes without extensions
aspectClasses = getAspectClasses(extensions=FALSE)

## set default updateClasses
updateAspectClasses(
  aspectClasses = aspectClasses
)

## default sub aspect classes
subAspectClasses
Description

This function creates a cartesian layout aspect, that stores coordinates of nodes.

Usage

createCartesianLayout(node, x, y, z = NULL, view = NULL)

Arguments

- **node**: integer; reference to node ids
- **x**: numeric; x coordinate
- **y**: numeric; y coordinate
- **z**: numeric (optional); z coordinate
- **view**: integer (optional); reference to subnetwork id of type view (CyNetworkRelations)

Details

The layout of networks can be influenced by setting the `node` position manually. While `x` and `y` coordinates are mandatory, the `z` coordinates are optional and can, for example, be used to define the vertical stacking order of overlapping nodes.

Similar to Cytoscape [https://cytoscape.org/](https://cytoscape.org/), it is possible to define different views of the same network. The views itself are defined in CySubNetworks and CyNetworkRelations, and only referenced by a unique subnetwork id.

Value

`CartesianLayoutAspect` object

See Also

`updateCartesianLayout`

Examples

```r
## a minimal example
cartesianLayout = createCartesianLayout(
  node=0,
  x=5.5,
  y=200.3
)

## defining several coordinates at once
cartesianLayout = createCartesianLayout(
```
## with all parameters

cartesianLayout = createCartesianLayout(
    node=c(0, 1, 0),
    x=c(5.5, 110.1, 7.2),
    y=c(200.3, 210.2, 13.9),
    z=c(-1, 3.1, NA),
    view=c(NA, NA, 1476)
)

---

<table>
<thead>
<tr>
<th>checks</th>
<th>Checks</th>
</tr>
</thead>
</table>

**Description**

Different functions to check parameters, ids, elements and lists

**Usage**

```
.paramClass(param, cls)
.checkClass(param, cls, name, cname = c())
.checkAllClass(L, cls, name, cname = c())
.checkClassOneOf(param, clss, name, cname = c())
.checkCharacter(param, name, cname = c())
.checkNumeric(param, name, cname = c())
.checkLogical(param, name, cname = c())
.checkList(param, name, cname = c())
.paramNamed(param)
.checkNamed(param, name, cname = c())
.checkNamedCharacter(param, name, cname = c())
.checkNamedNumeric(param, name, cname = c())
.checkNamedLogical(param, name, cname = c())
```
.checkNamedList(param, name, cname = c())
.paramNonNeg(param)
.checkNonNeg(param, name, cname = c())
.paramNoNa(param)
.checkNoNa(param, name, cname = c())
.checkIsUniqueId(param, name, cname = c())
.checkIsId(param, name, cname = c())
.paramIsOptionalId(param, name)
.checkSameLength(cname, ...)
.paramAnyNotNull(name, ...)
.checkAnyNotNull(name, cname = c(), ...)
.elementsUnique(A)
.checkUnique(A, name, cname = c())
.elementsUniqueDF(DF, cols)
.checkUniqueDF(DF, cols, cname = c())
.elementsInDict(A, key)
.elementsBContainsAllA(A, B)
.checkBContainsAllA(A, B, name, cname = c())
.checkRefs(A, B, name, cname = c())
.checkRefPresent(A, key, cls, name, cname = c())
.listAllNumeric(L)
.checkAllNumeric(L, name, cname = c())
.listAllNumericOrInDict(L, key)
.checkAllNumericOrInDict(L, key, name, cname = c())
Arguments

- **param**: some parameter.
- **cls**: character; class name.
- **name**: character; for logging the used name for the parameter.
- **cname**: character; for logging the name of the calling function.
- **L**: list.
- **clss**: character vector; list of class names.
- **...**: list of some vectors.
- **A, B**: vectors.
- **DF**: data.frame.
- **cols**: column names.
- **key**: name of the dictionary entry in .DICT.

Details

The .check* functions perform a test and stop with a custom error on fail. All other functions perform a test and return the result.

The used .DICT: looks as follows:

- VPpropertiesOf: network, nodes, edges, nodes:default, edges:default
- VPpropertyFields: properties, dependencies, mappings
- SN: all
- TCappliesTo: nodes, edges, networks

Value

logical

Functions

- .paramClass: checks if the object param is of class cls.
- .checkClass: checks if the object param is of class cls.
- .checkAllClass: checks if all elements of the list L are of class cls.
- .checkClassOneOf: checks if param is any class of clss.
- .checkCharacter: checks if param is character.
- .checkNumeric: checks if param is numeric.
- .checkLogical: checks if param is logical.
- .checkList: checks if param is a list.
- .paramNamed: checks if param has names.
- .checkNamed: checks if param has names.
- .checkNamedCharacter: checks if param has names and is character.
• .checkNamedNumeric: checks if `param` has names and is numeric.
• .checkNamedLogical: checks if `param` has names and is logical.
• .checkNamedList: checks if `param` has names and is a list.
• .paramNonNeg: checks if `param` is greater than 0 if not NA.
• .checkNonNeg: checks if `param` is greater than 0 if not NA.
• .paramNoNa: checks if `param` is not NA.
• .checkNoNa: checks if `param` is not NA.
• .checkIsUniqueId: checks if `param` is an unique id.
• .checkIsId: checks if `param` is an id.
• .paramIsOptionalId: checks if `param` is an optional id.
• .checkSameLength: checks if all elements in `...` have the same number of elements.
• .paramAnyNotNull: checks if any element in `...` is not NULL.
• .checkAnyNotNull: checks if any element in `...` is not NULL.
• .elementsUnique: checks if the elements in `A` are unique.
• .checkUnique: checks if the elements in `A` are unique.
• .elementsUniqueDF: checks if the elements in the columns `cols` of `DF` are unique.
• .checkUniqueDF: checks if the elements in the columns `cols` of `DF` are unique.
• .elementsInDict: checks if the elements of `A` are in `.DICT[[key]]`.
• .elementsBContainsAllA: checks if all elements of `A` are present in `B`.
• .checkBContainsAllA: checks if all elements of `A` are present in `B`.
• .checkRefs: checks if `B` contains all elements of `A`, aka. references.
• .checkRefPresent: checks if a referred aspect of class `cls` is accessible by `key` in `A`.
• .listAllNumeric: checks if all elements of a list `L` are numeric.
• .checkAllNumeric: checks if all elements of a list `L` are numeric.
• .listAllNumericOrInDict: checks if all elements of a list `L` are numeric or in `.DICT[[key]]`.
• .checkAllNumericOrInDict: checks if all elements of a list `L` are numeric or in `.DICT[[key]]`.

**Note**

Internal function only for convenience

**Examples**

```r
NULL
```
Description

Convert data types in `data.frame(dataType, isList)` to character of NDEx data types

Usage

```r
.convertDataTypes(df, cols = c(dataType = "dataType", isList = "isList"))
.convertValues(df, cols = c(value = "value", isList = "isList"))
```

Arguments

- **df**: data.frame with dataType and isList: `data.frame(dataType, isList)`
- **cols**: named character; column names of dataType and isList in df

Value

character; NDEx data types (e.g. "string" or "list_of_integer")

Note

Internal function only for convenience

Examples

```r
df = data.frame(dataType=c("string","boolean","double","integer","long",
                        "string","boolean","double","integer","long"),
                isList=c(FALSE,FALSE,FALSE,FALSE,FALSE,
                         TRUE,TRUE,TRUE,TRUE,TRUE))
df$value = list("string",TRUE,3.14,314,314,
               c("str","ing"),c(TRUE,FALSE),c(3.14,1.0),c(314,666),c(314,666))
RCX::.convertDataTypes(df)
RCX::.convertValues(df)
```
Description

The aspects in an RCX object are accessed by a name and return the aspect as an object of cls. To simplify the conversion between those, these functions return the corresponding name.

Usage

```r
aspectName2Class(name)

aspectClass2Name(cls)
```

Arguments

- name: character; name of the RCX accession of the Aspect
- cls: character; name of the aspect class

Details

The following accessions/classes are available within the standard RCX implementation:

```
accession name <=> class name

metaData <=> MetaDataAspect
nodes <=> NodesAspect
dges <=> EdgesAspect
nodeAttributes <=> NodeAttributesAspect
dgeAttributes <=> EdgeAttributesAspect
networkAttributes <=> NetworkAttributesAspect
cartesianLayout <=> CartesianLayoutAspect
cyGroups <=> CyGroupsAspect
cyVisualProperties <=> CyVisualPropertiesAspect
cyHiddenAttributes <=> CyHiddenAttributesAspect
cyNetworkRelations <=> CyNetworkRelationsAspect
cySubNetworks <=> CySubNetworksAspect
cyTableColumn <=> CyTableColumnAspect
```

Value

accession or class name
### Examples

```r
aspectName2Class("nodes")
## [1] "NodesAspect"

aspectClass2Name("NodesAspect")
## [1] "nodes"

aspectClasses

subAspectClasses
```

---

**convert2json**

*Convert data to json by R class*

**Description**

Convert data to json by R class

**Usage**

```r
.convert2json(x, ...)
```

- **.convert2json(x)**
- **.convert2json(x)**
- **.convert2json(x)**
- **.convert2json(x)**
- **.convert2json(x)**
- **.convert2json(x)**
- **.convert2json(x, raw = c(), byElement = FALSE, skipNa = TRUE)**
- **.convert2json(x, raw = c(), skipNa = TRUE)**

**Arguments**

- **x** data element
- **raw** character; names of columns not to format (e.g. because it is already converted)

**Value**

- character; json
countElements

Note

Internal function only for convenience

Examples

NULL

countElements | Number of elements in aspect

Description

This function returns the number of elements in an aspect.

Usage

countElements(x)

## Default S3 method:
countElements(x)

## S3 method for class 'RCX'
countElements(x)

## S3 method for class 'CyVisualPropertiesAspect'
countElements(x)

## S3 method for class 'MetaDataAspect'
countElements(x)

Arguments

x | an object of one of the aspect classes (e.g. Nodes) or RCX class.

Details

Uses method dispatch, so the default methods already returns the correct number for the most aspect classes. This way it is easier to extend the data model.

There are only two exceptions in the core and Cytoscape aspects: Meta-data and CyVisualProperties. Meta-data is a meta-aspect and therefore not included in Meta-data, and so its return is NA. CyVisualProperties is the only aspect with a complex data structure beneath. Therefore its number of elements is just the number of how many of the following properties are set: network, nodes, edges, defaultNodes or defaultEdges.
Value
integer; number of elements. For RCX objects all counts are returned in the vector named by the aspect class.

See Also
hasIds(), idProperty(), refersTo(), referredBy(), maxId()

Examples
nodes = createNodes(name = c("CDK1","CDK2","CDK3"))
edges = createEdges(source = c(0,0), target = c(1,2))
rcx = createRCX(nodes = nodes, edges = edges)
countElements(nodes)
countElements(rcx)

Description
These functions attempt to print RCX and aspect objects in a more readable form.

Usage
## S3 method for class 'MetaDataAspect'
print(x, ...)

## S3 method for class 'NodesAspect'
print(x, ...)

## S3 method for class 'EdgesAspect'
print(x, ...)

## S3 method for class 'NodeAttributesAspect'
print(x, ...)

## S3 method for class 'EdgeAttributesAspect'
print(x, ...)

## S3 method for class 'NetworkAttributesAspect'
print(x, ...)

## S3 method for class 'CartesianLayoutAspect'
print(x, ...)
## S3 method for class 'CyGroupsAspect'
print(x, ...)

## S3 method for class 'CyVisualPropertyProperties'
print(x, ...)

## S3 method for class 'CyVisualPropertyDependencies'
print(x, ...)

## S3 method for class 'CyVisualPropertyMappings'
print(x, ...)

## S3 method for class 'CyVisualProperty'
print(x, fields = c("all"), ...)

## S3 method for class 'CyVisualPropertiesAspect'
print(x, propertyOf = "all", fields = "all", ...)

## S3 method for class 'CyHiddenAttributesAspect'
print(x, ...)

## S3 method for class 'CyNetworkRelationsAspect'
print(x, ...)

## S3 method for class 'CySubNetworksAspect'
print(x, ...)

## S3 method for class 'CyTableColumnAspect'
print(x, ...)

## S3 method for class 'RCX'
print(x, unofficial = TRUE, ...)

**Arguments**

- **x**: aspect or RCX object
- **...**: further arguments passed to or from other methods. See *base::print()*
- **fields**: character; Which fields should be shown, one of properties, dependencies, mappings or all
- **propertyOf**: character; Which properties should be shown, one of network, nodes, edges, nodes:default, edges:default or all
- **inofficial**: logical; if FALSE only the official aspects are printed

**Value**

prints the object and returns it invisibly (*invisible*)
CyGroups

Cytoscape Groups

Description

This function is used to create Cytoscape "groups" aspects.

Usage

createCyGroups(
  id = NULL,
  name = NULL,
  nodes = NULL,
  externalEdges = NULL,
  internalEdges = NULL,
  collapsed = NULL
)

Arguments

id integer (optional); Cytoscape group ids
name character; names of the groups
nodes list of integers (optional); reference to node ids
externalEdges list of integers (optional); the external edges making up the group; reference to edge ids
internalEdges list of integers (optional); the internal edges making up the group; reference to edge ids
collapsed logical (optional); whether the group is displayed as a single node

Details

Cytoscape contributes aspects that organize subnetworks, attribute tables, and visual attributes for use by its own layout and analysis tools. Furthermore are the aspects used in web-based visualizations like within the NDEx platform.

Cytoscape groups allow to group a set of Nodes and corresponding internal and external Edges together, and represent a group as a single node in the visualization. A group is defined by its unique id, which must be an (positive) integer, which serves as reference to other aspects. If no ids are provided, they are created automatically.
CyHiddenAttributes

Value

`CyGroupsAspect` object

See Also

`updateCyGroups`;

Examples

```r
## a minimal example
cyGroups = createCyGroups(
  name = "Group One",
  nodes = list(c(1,2,3)),
  internalEdges = list(c(0,1))
)

## defining several groups at once
cyGroups = createCyGroups(
  name = c("Group One", "Group Two"),
  nodes = list(c(1,2,3), 0),
  internalEdges = list(c(0,1),NA)
)

## with all parameters
cyGroups = createCyGroups(
  id = c(0,1),
  name = c("Group One", "Group Two"),
  nodes = list(c(1,2,3), 0),
  internalEdges = list(c(0,1),NA),
  externalEdges = list(NA,c(1,3)),
  collapsed = c(TRUE,NA)
)
```

CyHiddenAttributes  Cytoscape hidden attributes

Description

This function is used to create Cytoscape hidden attributes aspects.

Usage

```r
createCyHiddenAttributes(
  name,
  value,
  dataType = NULL,
  isList = NULL,
  subnetworkId = NULL
)
```
Arguments

- **name**: character; key of the attribute
- **value**: character or list of character; value of the attribute
- **dataType**: character (optional); data type of the attribute
- **isList**: logical (optional); a value should be considered as list
- **subnetworkId**: integer (optional); refers to the IDs of a subnetwork aspect, but left blank (or NA) if root-network

Details

Cytoscape contributes aspects that organize subnetworks, attribute tables, and visual attributes for use by its own layout and analysis tools. Furthermore, these aspects are used in web-based visualizations like within the NDEx platform.

Besides network attributes, networks may have additional describing attributes originated from and used by Cytoscape. They are also defined in a key-value like manner, with the name of the attribute as key. The same attribute can also be defined for different subnetworks with different values. The values itself may differ in their data types, therefore it is necessary to provide the values as a list of the single values instead of a vector.

With **isList** it can be set, if a value should be considered as a list. This is of minor significance while working solely with RCX objects, unless it will be transformed to JSON. For some attributes it might be necessary that the values are encoded as lists, even if they contain only one element (or even zero elements). To force an element to be encoded correctly, this parameter can be used, for example:

```r
name = "A", value = "a", isList = T
```

will be encoded in JSON as `A=["a"]`.

Value

*CyHiddenAttributesAspect* object

See Also

- `updateCyHiddenAttributes`

Examples

```r
## a minimal example
hiddenAttributes = createCyHiddenAttributes(
    name="A",
    value="a"
)

## defining several properties at once
hiddenAttributes = createCyHiddenAttributes(
    name=c("A", "B"),
    value=c("a", "b")
)

## with characters and numbers mixed
hiddenAttributes = createCyHiddenAttributes(
```
CyNetworkRelations

Cytoscape network relations

### Description

This function is used to create Cytoscape network relations aspects.

```r
name=c("A","B"),
value=list("a",3.14)
)

## force the number to be characters
hiddenAttributes = createCyHiddenAttributes(
  name=c("A","B"),
  value=list("a",3.14),
  dataType=c("character","character")
)

## with a list as input for one value
hiddenAttributes = createCyHiddenAttributes(
  name=c("A","B"),
  value=list(c("a1","a2"),
            "b")
)

## force "B" to be a list as well
hiddenAttributes = createCyHiddenAttributes(
  name=c("A","B"),
  value=list(c("a1","a2"),
            "b"),
  isList=c(TRUE,TRUE)
)

## with a subnetwork
hiddenAttributes = createCyHiddenAttributes(
  name=c("A","A"),
  value=c("a","a with subnetwork"),
  subnetworkId=c(NA,1)
)

## with all parameters
hiddenAttributes = createCyHiddenAttributes(
  name=c("A","A","B","B"),
  value=list(c("a1","a2"),
            "a with subnetwork",
            "b",
            "b with subnetwork"),
  isList=c(TRUE,FALSE,TRUE,FALSE),
  subnetworkId=c(NA,1,NA,1)
)`
Usage

createCyNetworkRelations(child, parent = NULL, name = NULL, isView = FALSE)

Arguments

child    integer; reference to subnetwork id
parent   integer (optional); reference to subnetwork id, but left blank (or NA) for root-
          network
name     character (optional); name of the subnetwork or view
isView   logical (optional); TRUE for views, else the network defines a subnetwork

Details

Cytoscape contributes aspects that organize subnetworks, attribute tables, and visual attributes for
use by its own layout and analysis tools. Furthermore are the aspects used in web-based visualiza-
tions like within the NDEx platform.

Cytoscape network relations define the relationship between the main network, subnetworks and
views and also a name can be assigned to the relationship. Both, subnetworks and views are de-

dined as subnetworks aspect, but their type is defined here by the isView property. The parent of a
subnetwork or view can be an other subnetwork or the root network.

Value

CyNetworkRelationsAspect object

See Also

updateCyNetworkRelations;

Examples

## a minimal example
cyNetworkRelations = createCyNetworkRelations(
  child = 1
)

## with all parameters
cyNetworkRelations = createCyNetworkRelations(
  child = c(1,2),
  parent = c(NA,1),
  name = c("Network A", "View A"),
  isView = c(FALSE, TRUE)
)
CySubNetworks

Description
This function is used to create Cytoscape subnetwork aspects.

Usage
createCySubNetworks(id, nodes = NULL, edges = NULL)

Arguments
id    integer; subnetwork IDs
nodes integer; reference to node id OR character "all" to refer to all nodes
edges integer; reference to edge id OR character "all" to refer to all edges

Details
Cytoscape contributes aspects that organize subnetworks, attribute tables, and visual attributes for use by its own layout and analysis tools. Furthermore are the aspects used in web-based visualizations like within the NDEx platform.
A group is defined by its unique id, which must be an (positive) integer, which serves as reference to other aspects. If no IDs are provided, they are created automatically.
Nodes and edges are referred by the IDs of the corresponding aspect. Unlike other aspects referring those IDs, the Cytoscape subnetwork aspect allows to refer to all nodes and edges using the keyword all.
The relationship between (sub-)networks and views, and also the type (subnetwork or view) is defined in CyNetworkRelations.

Value
CySubNetworksAspect object

See Also
updateCySubNetworks;

Examples

```r
## a minimal example
cySubNetworks = createCySubNetworks(
    nodes = "all",
    edges = "all"
)

## defining several subnetworks at once
```
cySubNetworks = createCySubNetworks(
    nodes = list("all",
                c(1,2,3)),
    edges = list("all",
                c(0,2))
)

## with all parameters
cySubNetworks = createCySubNetworks(
    id = c(0,1),
    nodes = list("all",
                c(1,2,3)),
    edges = list("all",
                c(0,2))
)

---

**CyTableColumn**  
*Cytoscape table column properties*

**Description**

This function is used to create Cytoscape table column aspects.

**Usage**

```r
createCyTableColumn(
    appliesTo,  
    name,       
    dataType = NULL,  
    isList = NULL,  
    subnetworkId = NULL
)
```

**Arguments**

- **appliesTo**: character; indicates whether this applies to "nodes", "edges" or "networks" table columns
- **name**: character; key of the attribute
- **dataType**: character (optional); data type of the attribute
- **isList**: logical (optional); a value should be considered as list
- **subnetworkId**: integer (optional); reference to subnetwork id, but left blank (or NA) if root-network
CyVisualProperties

Details

Cytoscape contributes aspects that organize subnetworks, attribute tables, and visual attributes for use by its own layout and analysis tools. Furthermore are the aspects used in web-based visualizations like within the NDEx platform.

These elements are used to represent Cytoscape table column labels and types. Its main use is to disambiguate empty table columns. The same attribute can also be defined for different subnetworks with different values. Cytoscape does not currently support table columns for the root network, but this is option is included here for consistency.

With isList it can be set, if a value should be considered as a list. This is of minor significance while working solely with RCX objects, unless it will be transformed to JSON.

Value

`CyTableColumnAspect` object

See Also

`updateCyTableColumn`; `CyNetworkRelations`

Examples

```r
## a minimal example
tableColumn = createCyTableColumn(
  appliesTo="nodes",
  name="weight"
)

## defining several properties at once
tableColumn = createCyTableColumn(
  appliesTo=c("nodes","edges"),
  name=c("weight","weight")
)

## with all parameters
tableColumn = createCyTableColumn(
  appliesTo=c("nodes","edges","networks"),
  name=c("weight","weight","collapsed"),
  dataType=c("numeric","numeric","logical"),
  isList=c(FALSE,FALSE,TRUE),
  subnetworkId=c(NA,NA,1)
)
```

---

CyVisualProperties  Cytoscape visual properties (aspect)

Description

This function is used to create Cytoscape visual properties aspects, that consists of `CyVisualProperty` objects for networks, nodes, edges, and default nodes and edges.
Usage

createCyVisualProperties(
    network = NULL,
    nodes = NULL,
    edges = NULL,
    defaultNodes = NULL,
    defaultEdges = NULL
)

Arguments

- network: CyVisualProperty object (optional); the visual properties of networks
- nodes: CyVisualProperty object (optional); the visual properties of nodes
- edges: CyVisualProperty object (optional); the visual properties of edges
- defaultNodes: CyVisualProperty object (optional); the default visual properties of nodes
- defaultEdges: CyVisualProperty object (optional); the default visual properties of edges

Details

Cytoscape contributes aspects that organize subnetworks, attribute tables, and visual attributes for use by its own layout and analysis tools. Furthermore are the aspects used in web-based visualizations like within the NDEx platform.

The visual properties aspect is the only aspect (CyVisualProperties) with a complex structure. It is composed of several sub-property classes and consists of CyVisualProperty objects, that belong to, or more precisely describe one of the following network elements: network, nodes, edges, defaultNodes or defaultEdges.

A single visual property (i.e. CyVisualProperty object) organizes the information as properties, dependencies and mappings, as well as the single values appliesTo and view, that define the subnetwork or view to which the IDs apply.

Properties are CyVisualPropertyProperties objects, that hold information like "NODE_FILL_COLOR" : "#26CCC9" or "NODE_LABEL_TRANSPARENCY" : "255" in a key-value like manner.

Dependencies are CyVisualPropertyDependencies objects, that hold information about dependencies between visual properties. Currently there are only three dependencies supported:

- Lock Node with and height: nodeSizeLocked = "false"
- Fit Custom Graphics to node: nodeCustomGraphicsSizeSync = "true"
- Edge color to arrows: arrowColorMatchesEdge = "false"

Mappings are CyVisualPropertyMappings objects, that hold information as a triplet consisting of name, type and definition, like "NODE_FILL_COLOR" : "DISCRETE" : "COL=molecule_type,T=string,K=0=miRNA,V=0=#FCEC00" or "NODE_FILL_COLOR" : "CONTINUOUS" : "COL=gal1RGexp,T=double... or "NODE_LABEL" : "PASSTHROUGH" : "COL=COMMON,T=string".


Structure of Cytoscape Visual Properties:
CyVisualProperties

|---network = CyVisualProperty
|---nodes = CyVisualProperty
|---edges = CyVisualProperty
|---defaultNodes = CyVisualProperty
|---defaultEdges = CyVisualProperty

CyVisualProperty
|---properties = CyVisualPropertyProperties
  | |--name
  | |--value
|---dependencies = CyVisualPropertyDependencies
  | |--name
  | |--value
|---mappings = CyVisualPropertyMappings
  | |--name
  | |--type
  | |--definition
|---appliesTo = <reference to subnetwork id>
|---view = <reference to subnetwork id>

Value

CyVisualPropertiesAspect object

See Also

updateCyVisualProperties, updateCyVisualProperty, getCyVisualProperty

Examples

```r
## Prepare used properties
## Visual property: Properties
vpPropertyP1 = createCyVisualPropertyProperties(c(NODE_BORDER_STROKE="SOLID"))

## Visual property: Dependencies
vpPropertyD1 = createCyVisualPropertyDependencies(c(nodeSizeLocked="false"))

## Visual property: Mappings
vpPropertyM1 = createCyVisualPropertyMappings(c(NODE_FILL_COLOR="CONTINUOUS"),
                                           "COL=directed,T=boolean,K=0=true,V=0=ARROW")

## Create visual property object
vpProperty1 = createCyVisualProperty(properties=vpPropertyP1,
                                      dependencies=vpPropertyD1,
                                      mappings=vpPropertyM1)

## Create a visual properties aspect
## (using the same visual property object for simplicity)
createCyVisualProperties(network=vpProperty1,
                          nodes=vpProperty1,
```
CyVisualProperty

Cytoscape visual property (object used in CyVisualProperties aspect)

Description

This function is used to create Cytoscape visual property objects, that define networks, nodes, edges, and default nodes and edges in a CyVisualProperties aspect.

Usage

createCyVisualProperty(
  properties = NULL,
  dependencies = NULL,
  mappings = NULL,
  appliesTo = NULL,
  view = NULL
)

Arguments

properties       a single or a list of CyVisualPropertyProperties object (optional);
dependencies     a single or a list of CyVisualPropertyDependencies object (optional);
mappings         a single or a list of CyVisualPropertyMappings object (optional);
appliesTo        integer (optional); might refer to the IDs of a subnetwork aspect, but CX documentation is unclear
view             integer (optional); might refer to the IDs of a subnetwork aspect that is a view, but CX documentation is unclear

Details

Cytoscape contributes aspects that organize subnetworks, attribute tables, and visual attributes for use by its own layout and analysis tools. Furthermore are the aspects used in web-based visualizations like within the NDEx platform.

The visual properties aspect is the only aspect (CyVisualProperties) with a complex structure. It is composed of several sub-property classes and consists of CyVisualProperty objects, that belong to, or more precisely describe one of the following network elements: network, nodes, edges, defaultNodes or defaultEdges.

A single visual property (i.e. CyVisualProperty object) organizes the information as properties, dependencies and mappings, as well as the single values appliesTo and view, that define the subnetwork or view to which the IDs apply.

Properties are CyVisualPropertyProperties objects, that hold information like "NODE_FILL_COLOR": "#26CCC9" or "NODE_LABEL_TRANSPARENCY": "255" in a key-value like manner.
Dependencies are CyVisualPropertyDependencies objects, that hold information about dependencies between visual properties. Currently there are only three dependencies supported:

- Lock Node with and height: nodeSizeLocked = "false"
- Fit Custom Graphics to node: nodeCustomGraphicsSizeSync = "true"
- Edge color to arrows: arrowColorMatchesEdge = "false"

Mappings are CyVisualPropertyMappings objects, that hold information as a triplet consisting of name, type and definition, like "NODE_FILL_COLOR" : "DISCRETE" : "COL=molecule_type,T=string,K=0=miRNA,V=0=#FCEC00"


Structure of Cytoscape Visual Properties:

```plaintext
CyVisualProperties
|---network = CyVisualProperty
|---nodes = CyVisualProperty
|---edges = CyVisualProperty
|---defaultNodes = CyVisualProperty
|---defaultEdges = CyVisualProperty

CyVisualProperty
|---properties = CyVisualPropertyProperties
| |---name
| |---value
|---dependencies = CyVisualPropertyDependencies
| |---name
| |---value
|---mappings = CyVisualPropertyMappings
| |---name
| |---type
| |---definition
|---appliesTo = <reference to subnetwork id>
|---view = <reference to subnetwork id>
```

Value
---

CyVisualProperty object

See Also
---

updateCyVisualProperty, updateCyVisualProperties

Examples
---

```r
## Prepare used properties
## Visual property: Properties
vpPropertyNamedValue = c(NODE_BORDER_STROKE="SOLID",
```
## Visual property: Dependencies
vpDependencyNamedValue = c(nodeSizeLocked="false",
                           arrowColorMatchesEdge="true")
vpPropertyD = createCyVisualPropertyDependencies(vpDependencyNamedValue)

## Visual property: Mappings
vpMappingNamedType = c(NODE_FILL_COLOR="CONTINUOUS",
                        EDGE_TARGET_ARROW_SHAPE="DISCRETE")
vpMappingDefinition = c("COL=gal1RGexp,T=double,...",
                        "COL=directed,T=boolean,K=0=true,V=0=ARROW")
vpPropertyM = createCyVisualPropertyMappings(vpMappingNamedType,
                                             vpMappingDefinition)

## Create visual property object
createCyVisualProperty(properties=vpPropertyP,
                          dependencies=vpPropertyD,
                          mappings=vpPropertyM)

## Create visual property object with different subnetworks
createCyVisualProperty(properties=list(vpPropertyP,
                                       vpPropertyP),
                          dependencies=list(vpPropertyD,
                                            NA),
                          mappings=list(NA,
                                        vpPropertyM),
                          appliesTo = c(NA,
                                        1),
                          view = c(1,
                                   NA))

---

**CyVisualPropertyDependencies**

Create a object for dependency of Cytoscape Visual Properties (object used in CyVisualProperty)

---

### Description

This function is used to create aspects for mappings in Cytoscape visual properties. Networks, nodes, edges, and default nodes and edges mappings are realized as `CyVisualProperty` objects, that each consist of properties (`CyVisualPropertyProperties` objects), dependencies (this here) and mappings (`CyVisualPropertyMappings` objects).

### Usage

```r
createCyVisualPropertyDependencies(value, name = NULL)
```
Arguments

value character or named character; value of the dependencies
name character (optional); name of the dependencies

Details

Cytoscape contributes aspects that organize subnetworks, attribute tables, and visual attributes for use by its own layout and analysis tools. Furthermore are the aspects used in web-based visualizations like within the NDEx platform.

The visual properties aspect is the only aspect (CyVisualProperties) with a complex structure. It is composed of several sub-property classes and consists of CyVisualProperty objects, that belong to, or more precisely describe one of the following network elements: network, nodes, edges, defaultNodes or defaultEdges.

A single visual property (i.e. CyVisualProperty object) organizes the information as properties, dependencies and mappings, as well as the single values appliesTo and view, that define the subnetwork or view to which the IDs apply.

Properties are CyVisualPropertyProperties objects, that hold information like "NODE_FILL_COLOR" : "#26CCC9" or "NODE_LABEL_TRANSPARENCY" : "255" in a key-value like manner.

Dependencies are CyVisualPropertyDependencies objects, that hold information about dependencies between visual properties. Currently there are only three dependencies supported:

- Lock Node with and height: nodeSizeLocked = "false"
- Fit Custom Graphics to node: nodeCustomGraphicsSizeSync = "true"
- Edge color to arrows: arrowColorMatchesEdge = "false"

Mappings are CyVisualPropertyMappings objects, that hold information as a triplet consisting of name, type and definition, like "NODE_FILL_COLOR" : "DISCRETE" : "COL=molecule_type,T=string,K=0=miRNA,V=0=#FCEC00"


Value

CyVisualPropertyDependencies object

Note

If name is not provided, the names(value) is used instead to infer the names.

See Also

updateCyVisualProperty, updateCyVisualProperties
Examples

## Using a named vector
```
vpDependencyNamedValue = c(nodeSizeLocked="false",
                           arrowColorMatchesEdge="true")
createCyVisualPropertyDependencies(vpDependencyNamedValue)
```

## Using two separate vectors
```
vpDependencyName = c("nodeSizeLocked",
                    "arrowColorMatchesEdge")
vpDependencyValue = c("false",
                     "true")
createCyVisualPropertyDependencies(vpDependencyValue,
                          vpDependencyName)
```

# Result for either:

<table>
<thead>
<tr>
<th></th>
<th>name</th>
<th>value</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>nodeSizeLocked</td>
<td>false</td>
</tr>
<tr>
<td>2</td>
<td>arrowColorMatchesEdge</td>
<td>true</td>
</tr>
</tbody>
</table>

CyVisualPropertyMappings

Create an object for mappings of Cytoscape Visual Properties (object used in CyVisualProperty)

Description

This function is used to create objects for mappings in Cytoscape visual properties. Networks, nodes, edges, and default nodes and edges mappings are realized as CyVisualProperty objects, that each consist of properties (CyVisualPropertyProperties objects), dependencies (CyVisualPropertyDependencies objects) and mappings (this here).

Usage

```
createCyVisualPropertyMappings(type, definition, name = NULL)
```

Arguments

- **type**: character or named character; value of the mappings
- **definition**: character; definitions of the mappings
- **name**: character (optional); names of the mappings

Details

Cytoscape contributes aspects that organize subnetworks, attribute tables, and visual attributes for use by its own layout and analysis tools. Furthermore are the aspects used in web-based visualizations like within the NDEx platform.

The visual properties aspect is the only aspect (CyVisualProperties) with a complex structure. It is composed of several sub-property classes and consists of CyVisualProperty objects, that
belong to, or more precisely describe one of the following network elements: network, nodes, edges, defaultNodes or defaultEdges.

A single visual property (i.e. CyVisualProperty object) organizes the information as properties, dependencies and mappings, as well as the single values appliesTo and view, that define the subnetwork or view to which the IDs apply.

Properties are CyVisualPropertyProperties objects, that hold information like "NODE_FILL_COLOR" : "#26CCC9" or "NODE_LABEL_TRANSPARENCY" : "255" in a key-value like manner.

Dependencies are CyVisualPropertyDependencies objects, that hold information about dependencies between visual properties. Currently there are only three dependencies supported:

- Lock Node with and height: nodeSizeLocked = "false"
- Fit Custom Graphics to node: nodeCustomGraphicsSizeSync = "true"
- Edge color to arrows: arrowColorMatchesEdge = "false"

Mappings are CyVisualPropertyMappings objects, that hold information as a triplet consisting of name, type and definition, like "NODE_FILL_COLOR" : "DISCRETE" : "COL=molecule_type,T=string,K=0=miRNA,V=0=#FCEC00"


Structure of Cytoscape Visual Properties:

CyVisualProperties
  |---network = CyVisualProperty
  |---nodes = CyVisualProperty
  |---edges = CyVisualProperty
  |---defaultNodes = CyVisualProperty
  |---defaultEdges = CyVisualProperty

CyVisualProperty
  |---properties = CyVisualPropertyProperties
    |---name
    |---value
  |---dependencies = CyVisualPropertyDependencies
    |---name
    |---value
  |---mappings = CyVisualPropertyMappings
    |---name
    |---type
    |---definition
    |---appliesTo = <reference to subnetwork id>
    |---view = <reference to subnetwork id>

Value

CyVisualPropertyMappings object
CyVisualPropertyProperties

Create a object for properties of Cytoscape Visual Properties (object used in CyVisualProperty)

Description

This function is used to create aspects for mappings in Cytoscape visual properties. Networks, nodes, edges, and default nodes and edges mappings are realized as CyVisualProperty objects, that each consist of properties (this here), dependencies (CyVisualPropertyDependencies objects) and mappings (CyVisualPropertyMappings objects).

Usage

createCyVisualPropertyProperties(value, name = NULL)
CyVisualPropertyProperties

Arguments

value character or named character; value of the property
name character (optional); name of the property

Details

Cytoscape contributes aspects that organize subnetworks, attribute tables, and visual attributes for use by its own layout and analysis tools. Furthermore are the aspects used in web-based visualizations like within the NDEx platform.

The visual properties aspect is the only aspect (CyVisualProperties) with a complex structure. It is composed of several sub-property classes and consists of CyVisualProperty objects, that belong to, or more precisely describe one of the following network elements: network, nodes, edges, defaultNodes or defaultEdges.

A single visual property (i.e. CyVisualProperty object) organizes the information as properties, dependencies and mappings, as well as the single values appliesTo and view, that define the subnetwork or view to which the IDs apply.

Properties are CyVisualPropertyProperties objects, that hold information like "NODE_FILL_COLOR" : 
"#26CCC9" or "NODE_LABEL_TRANSPARENCY" : 
"255" in a key-value like manner.

Dependencies are CyVisualPropertyDependencies objects, that hold information about dependencies between visual properties. Currently there are only three dependencies supported:

- Lock Node with and height: nodeSizeLocked = "false"
- Fit Custom Graphics to node: nodeCustomGraphicsSizeSync = "true"
- Edge color to arrows: arrowColorMatchesEdge = "false"

Mappings are CyVisualPropertyMappings objects, that hold information as a triplet consisting of name, type and definition, like "NODE_FILL_COLOR" : "DISCRETE": "COL=molecule_type,T=string,K=0=miRNA,V=0=#FCEC00"


Structure of Cytoscape Visual Properties:

CyVisualProperties
|--- network = CyVisualProperty
|--- nodes = CyVisualProperty
|--- edges = CyVisualProperty
|--- defaultNodes = CyVisualProperty
|--- defaultEdges = CyVisualProperty

CyVisualProperty
|--- properties = CyVisualPropertyProperties
 | |--- name
 | |--- value
 |--- dependencies = CyVisualPropertyDependencies
 | |--- name
Value

CyVisualPropertyProperties object

Note

If name is not provided, the names(value) is used instead to infer the names.

See Also

updateCyVisualProperty, updateCyVisualProperties

Examples

## Using a named vector
vpPropertyNamedValue = c(NODE_BORDER_STROKE="SOLID",
                         NODE_BORDER_WIDTH="1.5")
createCyVisualPropertyProperties(vpPropertyNamedValue)

## Using two separate vectors
vpPropertyName = c("NODE_BORDER_STROKE",
                  "NODE_BORDER_WIDTH")
vpPropertyValue = c("SOLID",
                   "1.5")
createCyVisualPropertyProperties(vpPropertyValue,
                                 vpPropertyName)

# Result for either:
#  name  value
# 1  NODE_BORDER_STROKE  SOLID
# 2  NODE_BORDER_WIDTH  1.5

Description

Tests for validating RCX objects and its aspects.
Usage

.test_RequiredColumnsPresent(aspect, columns, verbose = FALSE)
.test_ListRequiredColumnsPresent(aspect, columns, verbose = FALSE)
.test_AllowedColumnsPresent(aspect, columns, verbose = FALSE)
.test_ListAllowedColumnsPresent(aspect, columns, verbose = FALSE)
.test_NoMergeColumn(aspect, column, verbose = FALSE)
.test_AtLeastOneColumnPresent(aspect, columns, verbose = FALSE)
.test_AtLeastOneElementPresent(aspect, element, verbose = FALSE)
.test_OneNodePresent(nodesAspect, column, verbose = FALSE)
.test_IsUnique(aspect, column, verbose = FALSE)
.test_ListAllUnique(aspect, column, verbose = FALSE)
.test_IsUniqueInLists(aspect, column, verbose = FALSE)
.test_ListAllUniqueInLists(aspect, column, verbose = FALSE)
.test_IsLogical(aspect, column, verbose = FALSE)
.test_IsNumeric(aspect, column, verbose = FALSE)
.test_ElementIsNumeric(aspect, element, verbose = FALSE)
.test_IsCharacter(aspect, column, verbose = FALSE)
.test_ListAllCharacter(aspect, element, verbose = FALSE)
.test_IsList(aspect, column, verbose = FALSE)
.test_ElementIsList(aspect, element, verbose = FALSE)
.test_IsPos(aspect, column, verbose = FALSE)
.test_IsClass(x, cls, verbose = FALSE)
.test_IsNamedList(aspect, names, verbose = FALSE)
.test_IsCVPclass(x, cls, verbose = FALSE)
.test_ListOfCVPclass(x, cls, verbose = FALSE)
.test.ContainsNA(aspect, column, verbose = FALSE)
.test.ListAllContainsNA(aspect, element, verbose = FALSE)
.test.ListAllNumeric(aspect, column, verbose = FALSE)
.test.ListAllNumericOrInDict(aspect, column, dic, verbose = FALSE)
.test.ListAllOfClass(aspect, cls, verbose = FALSE)
.test.AspectExist(rcx, aspect, verbose = FALSE)
.test.IdsInAspect(ids, aspect, column, info = "", verbose = FALSE)
.test.ValuesInSet(aspect, column, set, ignoreNA = TRUE, verbose = FALSE)
.test.DataTypeColumn(aspect, column, verbose = FALSE)

Arguments

aspect  one RCX aspect
columns  character; list of columns
verbose  logical (default=FALSE); also log the results
column  character; column name
cls  character; class name in .CLS or .CLSvp
names  character; names of list
dic  character; key in .DICT
rcx  RCX object
ids  numeric; ids
info  character (default=""); additional message for verbose
ignoreNA  logical (default=TRUE); ignore NA values

Value

logical; pass or fail the test

Functions

• .test.RequiredColumnsPresent: checks if aspect has all required columns
• .test.ListRequiredColumnsPresent: checks if all list elements have all required columns
• .test.AllowedColumnsPresent: checks if only allowed columns are set
• .test.ListAllowedColumnsPresent: checks if all list elements have only allowed columns
• .test.NoMergeColumn: checks if column with old ids is not present (would be a merge artefact)
dot_test

- `.test_AtLeastOneColumnPresent`: checks if at least one specified column is present
- `.test_AtLeastOneElementPresent`: checks if at least one specified element is present
- `.test_OneNodePresent`: checks if at least one element (node) is present in the specified column
- `.test_IsUnique`: checks if all elements in specified column are unique
- `.test_ListAllUnique`: checks for all list elements if all elements in specified column are unique
- `.test_IsUniqueInLists`: checks if all elements in specified column are unique
- `.test_ListAllUniqueInLists`: checks if all elements in specified column are unique
- `.test_IsLogical`: checks if the specified column is of type logical
- `.test_IsNumeric`: checks if the specified column is of type numeric
- `.test_ElementIsNumeric`: checks if the specified column is of type numeric
- `.test_IsCharacter`: checks if the specified column is of type character
- `.test_ListAllCharacter`: checks if the specified list element are all of type character
- `.test_IsList`: checks if the specified column is of type list
- `.test_ElementIsList`: checks if the specified column is of type list
- `.test_IsPos`: checks if the specified column are positive integers
- `.test_IsClass`: checks if the specified column is of the specified class in .CLS
- `.test_IsNamedList`: checks if the aspect is a list with specified names
- `.test_IsCVPclass`: checks if the specified column is of the specified class in .CLSvp
- `.test_ListOfCVPclass`: checks if the all elements in the list are of class in .CLSvp
- `.test_ContainsNA`: checks if the specified column contains any NA values
- `.test_ListAllContainsNA`: checks if the specified list element contains any NA values
- `.test_ListAllNumeric`: checks if the specified column is a list with only numeric values (NAs and NULLs are not considered)
- `.test_ListAllNumericOrInDict`: checks if the specified column is a list with only numeric values (NAs and NULLs are not considered) or in .DICT
- `.test_ListAllOfClass`: checks if the specified column is a list with only numeric values (NAs and NULLs are not considered) or in .DICT
- `.test_AspectExist`: checks if the rcx object contains the specified aspect
- `.test_IdsInAspect`: checks if all provided ids are present in the specified column of an aspect
- `.test_ValuesInSet`: checks if the specified column of an aspect only contains values of the provided set
- `.test_DataTypeColumn`: checks if the dataType column of an aspect only contains JSON data types.

**Note**

Internal function only for convenience
**EdgeAttributes**

**Edge attributes**

**Description**

This function creates an aspect for additional attributes of edges.

**Usage**

```r
createEdgeAttributes(
  propertyOf,  
  name,        
  value,       
  dataType = NULL, 
  isList = NULL, 
  subnetworkId = NULL 
)
```

**Arguments**

- `propertyOf` integer; reference to edge ids
- `name` character; key of the attribute
- `value` character; value of the attribute
- `dataType` character (optional); data type of the attribute
- `isList` logical (optional); a value should be considered as list
- `subnetworkId` integer (optional); reference to subnetwork id

**Details**

Edges may have additional attributes besides a name and a representation. Those additional attributes reference a edge by its id and are defined in a key-value like manner, with the name of the attribute as key. The same attribute can also be defined for different **subnetworks** with different values. The values itself may also differ in their data types, therefore it is necessary to provide the values as a list of the single values instead of a vector.

With `isList` it can be set, if a value should be considered as a list. This is of minor significance while working solely with **RCX** objects, unless it will be transformed to JSON. For some attributes it might be necessary that the values are encoded as lists, even if they contain only one element (or even zero elements). To force an element to be encoded correctly, this parameter can be used, for example: `name="A", value=a, isList=T` will be encoded in JSON as `A=["a"]`.

**Value**

*EdgeAttributesAspect* object
**Note**

The `propertyOf` parameter references the edge ids to which the attributes belong to. When adding an `EdgeAttributes` object to an RCX object, those ids must be present in the `Edges` aspect, otherwise an error is raised.

**See Also**

`updateEdgeAttributes`

**Examples**

```r
## a minimal example
edgeAttributes = createEdgeAttributes(
    propertyOf=1,
    name="A",
    value="a"
)

## defining several properties at once
edgeAttributes = createEdgeAttributes(
    propertyOf=c(1,1),
    name=c("A", "B"),
    value=c("a","b")
)

## with characters and numbers mixed
edgeAttributes = createEdgeAttributes(
    propertyOf=c(1,1),
    name=c("A","B"),
    value=list("a",3.14)
)

## force the number to be characters
edgeAttributes = createEdgeAttributes(
    propertyOf=c(1,1),
    name=c("A","B"),
    value=list("a",3.14),
    dataType=c("character","character")
)

## with a list as input for one value
edgeAttributes = createEdgeAttributes(
    propertyOf=c(1,1),
    name=c("A","B"),
    value=list(c("a1","a2"),
               "b")
)

## force "B" to be a list as well
eedgeAttributes = createEdgeAttributes(
    propertyOf=c(1,1),
    name=c("A","B"),
    value=list(c("a1","a2"),
               "b")
)
```
value=list(c("a1","a2"),
    "b"),
isList=c(TRUE,TRUE)
)

## with a subnetwork
edgeAttributes = createEdgeAttributes(
    propertyOf=c(1,1),
    name=c("A","A"),
    value=c("a","a with subnetwork"),
    subnetworkId=c(NA,1)
)

## with all parameters
edgeAttributes = createEdgeAttributes(
    propertyOf=c(1,1,1,1),
    name=c("A","A","B","B"),
    value=list(c("a1","a2"),
        "a with subnetwork",
        "b",
        "b with subnetwork"),
isList=c(TRUE,FALSE,TRUE,FALSE),
    subnetworkId=c(NA,1,NA,1)
)

---

## Edges

### Description

This function creates edges between nodes in networks.

### Usage

```r
createEdges(id = NULL, source, target, interaction = NULL)
```

### Arguments

- **id** integer (optional); edge IDs
- **source** integer; reference to node id
- **target** integer; reference to node id
- **interaction** character (optional); type of interaction, eg. "binds" or "activates"

### Details

Edges are represented by `EdgesAspect` objects. Edges connect two nodes, which means that `source` and `target` must reference the IDs of nodes in a `Nodes` object. On creation, the IDs don’t matter yet, but at least while adding the `EdgesAspect` object to an `RCX-object`, the IDs must be present in the nodes aspect of the `RCX-object`. 
Similar to nodes, an edge also has a unique \textit{id}, which must be an (positive) integer, which serves as reference to other aspects. If no IDs are provided, those are assigned automatically. Optionally, edges can have an interaction attribute to define the type of interaction between the nodes.

\textbf{Value}

\textit{EdgesAspect} object

\textbf{See Also}

\textit{updateEdges} for adding a \textit{EdgesAspect} object to an \textit{EdgesAspect} or \textit{RCX} object

\textbf{Examples}

```r
## create some simple edges
edges1 = createEdges(source=1, target=2)

## create edges with more information
edges2 = createEdges(id=c(3,2,4),
                     source=c(0,0,1),
                     target=c(1,2,2),
                     interaction=c("activates","inhibits", NA))
```

\textbf{getCyVisualProperty} \hspace{1cm} \textit{Get a Cytoscape visual property (object used in CyVisualProperties aspect) by appliesTo and view}

\textbf{Description}

This function helps filtering \textit{CyVisualProperty} objects by \textit{appliesTo} and \textit{view} attributes (i.e. a unique combination of both). If nothing matches the searched pattern \texttt{NULL} is returned.

\textbf{Usage}

```
getCyVisualProperty(cyVisualProperty, appliesTo = NA, view = NA)
```

\textbf{Arguments}

\begin{itemize}
  \item \texttt{cyVisualProperty} \hspace{1cm} \textit{CyVisualProperty} object
  \item \texttt{appliesTo} \hspace{1cm} integer (optional); value of \textit{appliesTo} to filter for
  \item \texttt{view} \hspace{1cm} integer (optional); value of \textit{view} to filter for
\end{itemize}
Details

Cytoscape contributes aspects that organize subnetworks, attribute tables, and visual attributes for use by its own layout and analysis tools. Furthermore are the aspects used in web-based visualizations like within the NDEx platform.

The visual properties aspect is the only aspect (CyVisualProperties) with a complex structure. It is composed of several sub-property classes and consists of CyVisualProperty objects, that belong to, or more precisely describe one of the following network elements: network, nodes, edges, defaultNodes or defaultEdges.

A single visual property (i.e. CyVisualProperty object) organizes the information as properties, dependencies and mappings, as well as the single values appliesTo and view, that define the subnetwork or view to which the IDs apply.

Properties are CyVisualPropertyProperties objects, that hold information like "NODE_FILL_COLOR" : ": #26CCC9" or "NODE_LABEL_TRANSPARENCY" : ": 255" in a key-value like manner.

Dependencies are CyVisualPropertyDependencies objects, that hold information about dependencies between visual properties. Currently there are only three dependencies supported:

- Lock Node with and height: nodeSizeLocked = "false"
- Fit Custom Graphics to node: nodeCustomGraphicsSizeSync = "true"
- Edge color to arrows: arrowColorMatchesEdge = "false"

Mappings are CyVisualPropertyMappings objects, that hold information as a triplet consisting of name, type and definition, like "NODE_FILL_COLOR" : "DISCRETE" : "COL=molecule_type,T=string,K=0=miRNA,V=0=#FCEC00" or "NODE_LABEL" : "CONTINUOUS" : "COL=molecule_type,T=string,K=0=data_value,V=0=#FCEC00".


Structure of Cytoscape Visual Properties:

CyVisualProperties
|---network = CyVisualProperty
|---nodes = CyVisualProperty
|---edges = CyVisualProperty
|---defaultNodes = CyVisualProperty
|---defaultEdges = CyVisualProperty

CyVisualProperty
|---properties = CyVisualPropertyProperties
| |---name
| |---value
|---dependencies = CyVisualPropertyDependencies
| |---name
| |---value
|---mappings = CyVisualPropertyMappings
| |---name
| |---type
| |---definition
getCyVisualProperty

---appliesTo = <reference to subnetwork id>
---view = <reference to subnetwork id>

Value

`CyVisualProperty` object containing only one element, or NULL

See Also

`updateCyVisualProperty`, `updateCyVisualProperties`

Examples

```r
## Visual property: Properties
vpPropertyP1 = createCyVisualPropertyProperties(c(NODE_BORDER_STROKE="SOLID"))

## Visual property: Dependencies
vpPropertyD1 = createCyVisualPropertyDependencies(c(nodeSizeLocked="false"))

## Visual property: Mappings
vpPropertyM1 = createCyVisualPropertyMappings(c(NODE_FILL_COLOR="CONTINUOUS"),
    "COL=directed,T=boolean,K=0=true,V=0=ARROW")

## Create visual property object
vpProperty = createCyVisualProperty(properties=list(vpPropertyP1, vpPropertyP1, vpPropertyP1),
    dependencies=list(vpPropertyD1, vpPropertyD1, NA),
    mappings=list(vpPropertyM1, NA, vpPropertyM1),
    appliesTo = c(NA, NA, 1),
    view = c(NA, 1, 1))

## Get VP for no subnetwork an no view
getCyVisualProperty(vpProperty)

getCyVisualProperty(vpProperty,
    appliesTo = 1,
    view = 1)
```
Description

Convert an RCX object to a graphNEL object

Usage

toGraphNEL(rcx, directed = FALSE)

fromGraphNEL(  
  graphNEL,  
  nodeId = "id",  
  nodeName = "nodeName",  
  nodeIgnore = c("name"),  
  edgeId = "id",  
  edgeInteraction = "edgeInteraction",  
  edgeIgnore = c(),  
  suppressWarning = FALSE  
)

Arguments

rcx               RCX object
directed          logical; whether the graph is directed
graphNEL           graphNEL object
nodeId             character; igraph attribute name used for node ids
nodeName           character; igraph attribute name used for node names
nodeIgnore         character; igraph attribute names that should be ignored
edgeId             character; igraph attribute name used for edge ids
edgeInteraction    character; igraph attribute name used for edge interaction
edgeIgnore         character; igraph attribute names that should be ignored
suppressWarning    logical; whether to suppress a warning message, if the validation of the RCX object fails

Details

In the graphNEL object the attributes are not separated from the graph like in RCX. Therefore, for converting an RCX object to a graphNEL object, and back, some adjustments in the naming of the attributes have to be made.

For nodes the name can be present in the nodes aspect, as name in the nodeAttributes aspect. Also name is used in graphNEL for naming the vertices. To avoid collisions in the conversion, the nodes
name is saved in graphNEL as nodeName, while the nodeAttributes property name is saved as "attribute...name". These names are also used for the conversion back to RCX, but here the name used in the nodes aspect can be changed by the nodeName parameter.

Similar to the node name, if "represents" is present as property in nodeAttributes its name is changed to "attribute...represents".

The conversion of edges works analogously: If "interaction" is present as property in edgeAttributes its name is changed to "attribute...interaction".

Nodes and edges must have IDs in the RCX, but not in the graphNEL object. To define an vertex or edge attribute to be used as ID, the parameters nodeId and edgeId can be used to define ether an attribute name (default:"id") or set it to NULL to generate ID automatically.

The attributes also may have a special data type assigned. The data type then is saved by adding "...dataType" to the attribute name.

The cartesian layout is also stored in the graphNEL object. To make those graph vertex attributes distinguishable from nodeAttributes they are named "cartesianLayout...x", "cartesianLayout...y" and "cartesianLayout...z".

In the RCX attributes it is also possible to define a subnetwork, to which an attribute applies. Those attributes are added with "...123" added to its name, where "123" is the subnetwork id. The subnetwork id itself are added as graph graph attributes, and are named subnetwork...123...nodes" and "subnetwork...123...edges", where "123" is the subnetwork id.

Altogether, the conventions look as follows: 

```
[attribute...]<name>[...<subnetwork>][...dataType]
```

Value

graphNEL or RCX object

See Also

igraph, igraph::as_graphnel()

Examples

```r
## Read from a CX file
## reading the provided example network of the package
cxFile <- system.file(
  "extdata",
  "Imatinib-Inhibition-of-BCR-ABL-66a902f5-2022-11e9-bb6a-0ac135e8bacf.cx",
  package = "RCX"
)

rcx = readCX(cxFile)

## graphNEL can handle multi-edges, but only if the graph is directed and the
## source and target start and end not between the same nodes.
## Unfortunately this is the case in our sample network.
## A quick fix is simply switching the direction of source and target
## for the multi-edges:
dubEdges = duplicated(rcx$edges[c("source","target")])
```
hasIds

IDs of an aspect

Description

This function checks if an aspect has IDs that may be referenced by other aspects.

By default aspects don't have IDs, so only the implemented classes have IDs. Aspects with IDs will be considered in the meta-data aspect to determine properties like: idCounter and elementCount.

Usage

hasIds(aspect)

## Default S3 method:
hasIds(aspect)

## S3 method for class 'NodesAspect'
hasIds(aspect)

## S3 method for class 'EdgesAspect'
hasIds(aspect)

## S3 method for class 'CyGroupsAspect'
hasIds(aspect)

## S3 method for class 'CySubNetworksAspect'
hasIds(aspect)

Arguments

aspect an object of one of the aspect classes (e.g. NodesAspect, EdgesAspect, etc.)

Details

Uses method dispatch, so the default return is FALSE and only aspect classes with IDs are implemented. This way it is easier to extend the data model.
idProperty

Value

logical

See Also

idProperty(), refersTo(), referredBy(), maxId()

Examples

edges <- createEdges(source = c(0,0), target = c(1,2))
hasIds(edges)

| idProperty | Name of the property of an aspect that is an ID |

Description

This function returns the name of the property, if an aspect uses IDs for its elements. As example, the aspect NodesAspect has the property id that represents the IDs of the aspect.

Usage

idProperty(aspect)

## Default S3 method:
idProperty(aspect)

## S3 method for class 'NodesAspect'
idProperty(aspect)

## S3 method for class 'EdgesAspect'
idProperty(aspect)

## S3 method for class 'CyGroupsAspect'
idProperty(aspect)

## S3 method for class 'CySubNetworksAspect'
idProperty(aspect)

Arguments

aspect an object of one of the aspect classes (e.g. NodesAspect, EdgesAspect, etc.)

Details

By default aspects don’t have IDs, so only the implemented classes have IDs. Aspects with IDs will be considered in the meta-data aspect to determine properties like: idCounter and elementCount.

Uses method dispatch, so the default return is NULL and only aspect classes with IDs are implemented. This way it is easier to extend the data model.
Value

character; Name of the ID property or NULL

See Also

hasIds(), refersTo(), referredBy(), maxId()

Examples

edges = createEdges(source = c(0,0), target = c(1,2))
idProperty(edges)

Convert an RCX object from and to an igraph object

Description

Convert an RCX object to an igraph object

Usage

```
toIgraph(rcx, directed = FALSE)
```

```
fromIgraph(
  ig,
  nodeId = "id",
  nodeName = "nodeName",
  nodeIgnore = c("name"),
  edgeId = "id",
  edgeInteraction = "edgeInteraction",
  edgeIgnore = c(),
  suppressWarning = FALSE
)
```

Arguments

- **rcx**  
  RCX object
- **directed**  
  logical; whether the graph is directed
- **ig**  
  igraph object
- **nodeId**  
  character; igraph attribute name used for node ids
- **nodeName**  
  character; igraph attribute name used for node names
- **nodeIgnore**  
  character; igraph attribute names that should be ignored
- **edgeId**  
  character; igraph attribute name used for edge ids
- **edgeInteraction**  
  character; igraph attribute name used for edge interaction
edgeIgnore character; igraph attribute names that should be ignored
suppressWarning logical; whether to suppress a warning message, if the validation of the RCX object fails

Details

In the igraph object the attributes are not separated from the graph like in RCX. Therefore, for converting an RCX object to an igraph object, and back, some adjustments in the naming of the attributes have to be made.

For nodes the name can be present in the nodes aspect, as name in the nodeAttributes aspect. Also name is used in igraph for naming the vertices. To avoid collisions in the conversion, the nodes name is saved in igraph as nodeName, while the nodeAttributes property name is saved as "attribute...name". These names are also used for the conversion back to RCX, but here the name used in the nodes aspect can be changed by the nodeName parameter.

Similar to the node name, if "represents" is present as property in nodeAttributes its name is changed to "attribute...represents".

The conversion of edges works analogously: If "interaction" is present as property in edgeAttributes its name is changed to "attribute...interaction".

Nodes and edges must have IDs in the RCX, but not in the igraph object. To define an vertex or edge attribute to be used as ID, the parameters nodeId and edgeId can be used to define ether an attribute name (default:"id") or set it to NULL to generate ID automatically.

The attributes also may have a special data type assigned. The data type then is saved by adding "...dataType" to the attribute name.

The cartesian layout is also stored in the igraph object. To make those igraph vertex attributes distinguishable from nodeAttributes they are named "cartesianLayout...x","cartesianLayout...y" and "cartesianLayout...z".

In the RCX attributes it is also possible to define a subnetwork, to which an attribute applies. Those attributes are added with "...123" added to its name, where "123" is the subnetwork id. The subnetwork id itself are added as igraph graph attributes, and are named subnetwork...123...nodes" and "subnetwork...123...edges", where "123" is the subnetwork id.

Altogether, the conventions look as follows: "[attribute...]<name>[...<subnetwork>][...dataType]"

Value

igraph or RCX object

See Also

graphNEL

Examples

## Read from a CX file
## reading the provided example network of the package
cxFile <- system.file( "extdata",
"Imatinib-Inhibition-of-BCR-ABL-66a902f5-2022-11e9-bb6a-0ac135e8bacf.cx",
package = "RCX"
)

rcx = readCX(cxFile)

## convert the network to igraph
ig = toIgraph(rcx)

## convert it back
rcxFromIg = fromIgraph(ig)

---

**jsonToRCX**  
*Convert parsed JSON aspects to RCX*

**Description**

Functions to handle parsed JSON for the different aspects.

**Usage**

jsonToRCX(jsonData, verbose)

---

**jsonToRCX**

## Default S3 method:
jsonToRCX(jsonData, verbose)

## S3 method for class 'status'
jsonToRCX(jsonData, verbose)

## S3 method for class 'numberVerification'
jsonToRCX(jsonData, verbose)

## S3 method for class 'metaData'
jsonToRCX(jsonData, verbose)

## S3 method for class 'nodes'
jsonToRCX(jsonData, verbose)

## S3 method for class 'edges'
jsonToRCX(jsonData, verbose)

## S3 method for class 'nodeAttributes'
jsonToRCX(jsonData, verbose)

## S3 method for class 'edgeAttributes'
jsonToRCX(jsonData, verbose)

## S3 method for class 'networkAttributes'
jsonToRCX(jsonData, verbose)

## S3 method for class 'cartesianLayout'
jsonToRCX(jsonData, verbose)

## S3 method for class 'cyGroups'
jsonToRCX(jsonData, verbose)

## S3 method for class 'cyHiddenAttributes'
jsonToRCX(jsonData, verbose)

## S3 method for class 'cyNetworkRelations'
jsonToRCX(jsonData, verbose)

## S3 method for class 'cySubNetworks'
jsonToRCX(jsonData, verbose)

## S3 method for class 'cyTableColumn'
jsonToRCX(jsonData, verbose)

## S3 method for class 'cyVisualProperties'
jsonToRCX(jsonData, verbose)

### Arguments

- **jsonData**: nested list from parsed JSON
- **verbose**: logical; whether to print what is happening

### Details

These functions will be used in processCX to process the JSON data for every aspect. Each aspect is accessible in the CX-JSON by a particular accession name (i.e. its aspect name; see NDEx documentation: [https://home.ndexbio.org/data-model/](https://home.ndexbio.org/data-model/)). This name is used as class to handle different aspects by method dispatch. This simplifies the extension of RCX for non-standard or self-defined aspects.

The CX-JSON is parsed to R data types using the jsonlite package as follows:

```r
jsonlite::fromJSON(cx, simplifyVector = FALSE)
```

This results in a list of lists (of lists...) to avoid automatic data type conversions, which affect the correctness and usability of the data. Simplified JSON data for example NodeAttributes would be coerced into a data.frame, therefore the value column looses the format for data types other than string.

The `jsonData` will be a list with only one element named by the aspect: `jsonData$<accessionName>`

To access the parsed data for example nodes, this can be done by `jsonData$nodes`. The single aspects are then created using the corresponding `create` functions and combined to an RCX object using the corresponding `update` functions.
markAttributeColumn

Mark attribute name columns within a data.frame

Description

Assigns a class to a data.frame column to force a custom format in summary generation.

Usage

.markAttributeColumn(aspect) <- value

Arguments

aspect an aspect (data.frame)
value character; property

Value

the aspect (data.frame)

Note

Internal function only for convenience

Examples

df = data.frame(name=c("a","b","c"),
               value=c("a","b","c"))
RCX:::markRefColumn(df) = "name"
summary(df)
**markRefColumn**

Mark required and optional references within a data.frame

---

**Description**

Assigns a class to a data.frame column to force a custom format in summary generation.

**Usage**

```
.markRefColumn(aspect) <- value
.markReqRefColumn(aspect) <- value
```

**Arguments**

- **aspect**: an aspect (data.frame)
- **value**: character; property

**Value**

the aspect (data.frame)

**Note**

Internal function only for convenience

**Examples**

```
df = data.frame(bla=c("a","b","c"),
       blubb=c("a","b","c"))
RCX:::markRefColumn(df) = "bla"
summary(df)
```

---

**maxId**

Highest ID of an aspect

---

**Description**

This function returns the highest id used in an aspect, that has ids. As example, the aspect *Node-Aspect* has the property *id* that must be a unique positive integer.
Usage

```r
maxId(x)
```

```r
## Default S3 method:
maxId(x)
```

```r
## S3 method for class 'RCX'
maxId(x)
```

Arguments

- `x`: an object of one of the aspect classes (e.g. NodesAspect, EdgesAspect, etc.) or `RCX` class.

Details

Uses method dispatch, so the default return is `NULL` and only aspect classes that have ids are implemented. This way it is easier to extend the data model.

Value

integer; Highest id. For `RCX` objects all highest ids are returned in the vector named by the aspect class.

See Also

`hasIds()`, `idProperty()`, `refersTo()`, `referredBy()`, `maxId()`

Examples

```r
nodes = createNodes(name = c("CDK1","CDK2","CDK3"))
maxId(nodes)
```

---

**Meta-data**

Update RCX meta-data

Description

The meta-data aspect contains meta-data about the aspects in the `RCX` object. It can be generated automatically based on the aspects present in a `RCX` object:

- for `version` and `consistencyGroup` default values are used
- `idCounter` is inferred with `hasIds` and `maxId` of an aspect
- `elementCount` is inferred from `countElements`
- `properties` is left out by default
Usage

updateMetaData(
  x,
  version = NULL,
  consistencyGroup = NULL,
  properties = NULL,
  aspectClasses = getAspectClasses()
)

## S3 method for class 'RCX'
updateMetaData(
  x,
  version = NULL,
  consistencyGroup = NULL,
  properties = NULL,
  aspectClasses = getAspectClasses()
)

## Default S3 method:
updateMetaData(
  x,
  version = NULL,
  consistencyGroup = NULL,
  properties = NULL,
  aspectClasses = getAspectClasses()
)

Arguments

x          RCX object or an aspect of a RCX; its class must be one of the standard RCX aspect classes
version    named character (optional); version of the aspect (default:"1.0")
consistencyGroup named numerical (optional); consistency group of the aspect (default:1)
properties  named list (optional); properties that need to be fetched or updated independently of aspect data
aspectClasses named character; accession names and aspect classes aspectClasses

Details

If version, consistencyGroup or properties should have a different value, they can be set using a named vector (or named list for properties), where the name must be an accession name of that aspect in the RCX-object (e.g. nodes or cyVisualProperties).

Besides being a named list by aspect accession name, properties must also contain the single key-value pairs as a further named list. To remove all key-value pairs for one aspect, an empty list can be provided instead of a list with key-value pairs. To simplify adding of properties to a single aspect, there is the updateMetaDataProperties function available.
NetworkAttributes

**Value**

MetaDataAspect object or RCX object

**Note**

The meta-data will always be updated automatically, when an aspect is added to or changed in the RCX object.

**See Also**

updateMetaDataProperties

**Examples**

```r
## prepare RCX object:
nodes = createNodes(name = c("a","b","c","d","e","f"))
edges = createEdges(source=c(1,2,0,0,0,2),
                   target=c(2,3,1,2,5,4))
rcx = createRCX(nodes, edges)
cySubNetworks = createCySubNetworks(
    id = c(1,2),
    nodes = list("all", c(1,2,3)),
    edges = list("all", c(0,2))
)
rcx = updateCySubNetworks(rcx, cySubNetworks)

## update meta-data manually
rcx = updateMetaData(rcx)

## update meta-data with some values
rcx = updateMetaData(rcx,
    version=c(edges="2.0"),
    consistencyGroup=c(nodes=3),
    properties=list(cySubNetworks=list(some="value",
                                       another="VALUE"),
                   edges=list(some="edge",
                               another="EDGE")))

## remove all properties for edges
rcx = updateMetaData(rcx, properties=list(edges=list()))
```

<table>
<thead>
<tr>
<th>NetworkAttributes</th>
<th>Network attributes</th>
</tr>
</thead>
</table>

**Description**

This function creates an aspect for attributes of a network.
createNetworkAttributes(
    name, 
    value, 
    dataType = NULL, 
    isList = NULL, 
    subnetworkId = NULL
)

Arguments

name character; key of the attribute
value character; value of the attribute
dataType character (optional); data type of the attribute
isList logical (optional); a value should be considered as list
subnetworkId integer (optional); reference to subnetwork id

Details

Networks may have describing attributes, that are defined in a key-value like manner, with the name of the attribute as key. The same attribute can also be defined for different subnetworks with different values. The values itself may differ in their data types, therefore it is necessary to provide the values as a list of the single values instead of a vector.

With isList it can be set, if a value should be considered as a list. This is of minor significance while working solely with RCX objects, unless it will be transformed to JSON. For some attributes it might be necessary that the values are encoded as lists, even if they contain only one element (or even zero elements). To force an element to be encoded correctly, this parameter can be used, for example: name="A", value=a, isList=T will be encoded in JSON as A=["a"].

Value

NetworkAttributesAspect object

See Also

updateNetworkAttributes; NodeAttributes, EdgeAttributes

Examples

## a minimal example
networkAttributes = createNetworkAttributes(
    name="A", 
    value="a"
)

## defining several properties at once
networkAttributes = createNetworkAttributes(
    name=c("A", "B"),
)
NodeAttributes

value=c("a","b")
)

## with characters and numbers mixed
networkAttributes = createNetworkAttributes(
    name=c("A","B"),
    value=list("a",3.14)
)

## force the number to be characters
networkAttributes = createNetworkAttributes(
    name=c("A","B"),
    value=list("a",3.14),
    dataType=c("character","character")
)

## with a list as input for one value
networkAttributes = createNetworkAttributes(
    name=c("A","B"),
    value=list(c("a1","a2"),
              "b")
)

## force "B" to be a list as well
networkAttributes = createNetworkAttributes(
    name=c("A","B"),
    value=list(c("a1","a2"),
              "b"),
    isList=c(TRUE,TRUE)
)

## with a subnetwork
networkAttributes = createNetworkAttributes(
    name=c("A","A"),
    value=c("a","a with subnetwork"),
    subnetworkId=c(NA,1)
)

## with all parameters
networkAttributes = createNetworkAttributes(
    name=c("A","A","B","B"),
    value=list(c("a1","a2"),
               "a with subnetwork",
               "b",
               "b with subnetwork"),
    isList=c(TRUE,FALSE,TRUE,FALSE),
    subnetworkId=c(NA,1,NA,1)
)
Description

This function creates an aspect for additional attributes of nodes.

Usage

createNodeAttributes(
  propertyOf,
  name,
  value,
  dataType = NULL,
  isList = NULL,
  subnetworkId = NULL
)

Arguments

propertyOf integer; reference to node ids
name character; key of the attribute
value character; value of the attribute
dataType character (optional); data type of the attribute
isList logical (optional); a value should be considered as list
subnetworkId integer (optional); reference to subnetwork id

Details

Nodes may have additional attributes besides a name and a representation. Those additional attributes reference a node by its id and are defined in a key-value like manner, with the name of the attribute as key. The same attribute can also be defined for different subnetworks with different values. The values itself may also differ in their data types, therefore it is necessary to provide the values as a list of the single values instead of a vector.

With isList it can be set, if a value should be considered as a list. This is of minor significance while working solely with RCX objects, unless it will be transformed to JSON. For some attributes it might be necessary that the values are encoded as lists, even if they contain only one element (or even zero elements). To force an element to be encoded correctly, this parameter can be used, for example: name="A", value=a, isList=T will be encoded in JSON as A=["a"].

Value

NodeAttributesAspect object

Note

The propertyOf parameter references the node ids to which the attributes belong to. When adding an NodeAttributesAspect object to an RCX object, those ids must be present in the Nodes aspect, otherwise an error is raised.
See Also

updateNodeAttributes, EdgeAttributes, NetworkAttributes

Examples

```r
## a minimal example
nodeAttributes = createNodeAttributes(
    propertyOf=1,
    name="A",
    value="a"
)

## defining several properties at once
nodeAttributes = createNodeAttributes(
    propertyOf=c(1,1),
    name=c("A", "B"),
    value=c("a","b")
)

## with characters and numbers mixed
nodeAttributes = createNodeAttributes(
    propertyOf=c(1,1),
    name=c("A","B"),
    value=list("a",3.14)
)

## force the number to be characters
nodeAttributes = createNodeAttributes(
    propertyOf=c(1,1),
    name=c("A","B"),
    value=list("a",3.14),
    dataType=c("string","string")
)

## with a list as input for one value
nodeAttributes = createNodeAttributes(
    propertyOf=c(1,1),
    name=c("A","B"),
    value=list(c("a1","a2"),
              "b")
)

## force "B" to be a list as well
nodeAttributes = createNodeAttributes(
    propertyOf=c(1,1),
    name=c("A","B"),
    value=list(c("a1","a2"),
               "b"),
    isList=c(TRUE,TRUE)
)

## with a subnetwork
```
Nodes

NodeAttributes = createNodeAttributes(
  propertyOf=c(1,1),
  name=c("A","A"),
  value=c("a","a with subnetwork"),
  subnetworkId=c(NA,1)
)

## with all parameters
NodeAttributes = createNodeAttributes(
  propertyOf=c(1,1,1,1,1,1),
  name=c("A","A","b","d","i","l"),
  value=list(c("a1","a2"),
             "a with subnetwork",
             TRUE,
             3.14,
             314,
             3144),
  dataType=c("string","string","boolean","double","integer","long"),
  isList=c(TRUE,FALSE,FALSE,FALSE,FALSE,FALSE),
  subnetworkId=c(NA,1,NA,NA,NA,NA)
)

Description

This function creates nodes for networks.

Usage

createNodes(id = NULL, name = NULL, represents = NULL)

Arguments

id integer (optional); node IDs
name character (optional); names of the nodes
represents character (optional); representation, e.g. a link to another database

Details

Nodes are represented by NodesAspect objects. A single node is defined by its unique id, which must be an (positive) integer, which serves as reference to other aspects. Optionally, nodes can have a name and a represents attribute. If no IDs are provided, but either names or representations (or both) IDs are assigned automatically. To be valid, a nodes aspect must contain at least one node. However, if no parameters are set (i.e. id, name and represents = NULL) there is still one node created with neither name nor representation, just an ID. The NodesAspect is the only mandatory aspect for an RCX-object.
Value

*NodesAspect* object

See Also

*updateNodes, RCX-object*

Examples

```r
## a minimal example
nodes = createNodes()

## ids will be generated
nodes = createNodes(name = c("a", "b", "c"))

## with all parameters
nodes = createNodes(id = c(1, 2, 3),
                    name = c("CDK1", "CDK2", "CDK3"),
                    represents = c("HGNC:CDK1",
                                    "Uniprot:P24941",
                                    "Ensembl:ENSG00000250506"))
```

RCX

*R package implementing the Cytoscape Exchange (CX) format*

Description

Create, handle, validate, visualize and convert networks in the Cytoscape exchange (CX) format to standard data types and objects.

Details

The CX format is also used by the NDEx platform, an online commons for biological networks, and the network visualization software Cytoscape.

browseVignettes("RCy3")

Author(s)

Florian Auer <florian.auer@informatik.uni-augsburg.de>
Description

An RCX object consists of several aspects, but at least one node in the nodes aspect. The network can either be created by creating every single aspect first and the create the network with all aspects present, or by creating the aspect only with the nodes and adding the remaining aspects one by one.

Usage

```r
createRCX(
  nodes,
  edges,
  nodeAttributes,
  edgeAttributes,
  networkAttributes,
  cartesianLayout,
  cyGroups,
  cyVisualProperties,
  cyHiddenAttributes,
  cyNetworkRelations,
  cySubNetworks,
  cyTableColumn,
  checkReferences = TRUE
)
```

Arguments

- `nodes` Nodes aspect;
- `edges` Edges aspect (optional);
- `nodeAttributes` NodeAttributes aspect (optional);
- `edgeAttributes` EdgeAttributes aspect (optional);
- `networkAttributes` NetworkAttributes aspect (optional);
- `cartesianLayout` CartesianLayout aspect (optional);
- `cyGroups` CyGroups aspect (optional);
- `cyVisualProperties` CyVisualProperties aspect (optional);
- `cyHiddenAttributes` CyHiddenAttributes aspect (optional);
- `cyNetworkRelations` CyNetworkRelations aspect (optional);
cySubNetworks  
CySubNetworks aspect (optional);

cyTableColumn  
CyTableColumn aspect (optional);

checkReferences

 logical; whether to check if references to other aspects are present in the RCX object

Details

vignette("01. RCX - an R package implementing the Cytoscape Exchange (CX) format", package = "RCX")
vignette("02. Creating RCX from scratch", package = "RCX")
vignette("Appendix: The RCX and CX Data Model", package = "RCX")

Value

RCX object

Examples

## minimal example
rcx = createRCX(createNodes())

## create by aspect
nodes = createNodes(name = c("a","b","c"))
edges = createEdges(source=c(0,0), target=c(1,2))

nodeAttributes = createNodeAttributes(
    propertyOf=c(1,1),
    name=c("A","B"),
    value=c("a","b")
)

edgeAttributes = createEdgeAttributes(
    propertyOf=c(0,0),
    name=c("A", "B"),
    value=c("a", "b")
)

networkAttributes = createNetworkAttributes(
    name=c("A","B"),
    value=list("a",3.14)
)

cartesianLayout = createCartesianLayout(
    node=c(0, 1),
    x=c(5.5, 110.1),
    y=c(200.3, 210.2)
)

cyGroups = createCyGroups(
    name = c("Group One", "Group Two"),
    nodes = list(c(0,1), 0)
vpPropertyP = createCyVisualPropertyProperties(c(NODE_BORDER_STROKE="SOLID"))
vpPropertyD = createCyVisualPropertyDependencies(c(nodeSizeLocked="false"),
  "COL=directed,T=boolean,K=0=true,V=0=ARROW")
vpPropertyM = createCyVisualPropertyMappings(c(NODE_FILL_COLOR="CONTINUOUS"),
  "COL=directed,T=boolean,K=0=true,V=0=ARROW")
vpProperty = createCyVisualProperty(properties=vpPropertyP,
  dependencies=vpPropertyD,
  mappings=vpPropertyM)

cyVisualProperties = createCyVisualProperties(nodes=vpProperty)

cyHiddenAttributes = createCyHiddenAttributes(
  name=c("A","B"),
  value=list(c("a1","a2"), "b")
)

cyNetworkRelations = createCyNetworkRelations(
  child = c(0,1),
  name = c("Network A", NA)
)

cySubNetworks = createCySubNetworks(
  nodes = list("all", c(0,1,2)),
  edges = list("all", c(0,1)))

cyTableColumn = createCyTableColumn(
  appliesTo=c("nodes","edges","networks"),
  name=c("weight","weight","collapsed"),
  dataType=c("double","double","boolean")
)

crx = createRCX(nodes, edges,
  nodeAttributes, edgeAttributes,
  networkAttributes,
  cartesianLayout,
  cyGroups,
  cyVisualProperties,
  cyHiddenAttributes,
  cyNetworkRelations,
  cySubNetworks,
  cyTableColumn)

## create all at once
rcx = createRCX(
  createNodes(name = c("a","b","c"),
  createEdges(source=c(0,0), target=c(1,2)),
  createNodeAttributes(
    propertyOf=c(1,1),
    name=c("A","B"),
    value=c("a","b"))
),
```r
createEdgeAttributes(
  propertyOf=c(0,0),
  name=c("A", "B"),
  value=c("a","b")
),

networkAttributes = createNetworkAttributes(
  name=c("A","B"),
  value=list("a",3.14)
),
cartesianLayout = createCartesianLayout(
  node=c(0, 1),
  x=c(5.5, 110.1),
  y=c(200.3, 210.2)
),
createCyGroups(
  name = c("Group One", "Group Two"),
  nodes = list(c(0,1), 0)
),
createCyVisualProperties(
  nodes=createCyVisualProperty(
    properties=createCyVisualPropertyProperties(
      c(NODE_BORDER_STROKE="SOLID")
    ),
    dependencies=createCyVisualPropertyDependencies(
      c(nodeSizeLocked="false")
    ),
    mappings=createCyVisualPropertyMappings(
      c(NODE_FILL_COLOR="CONTINUOUS"),
      "COL=directed,T=boolean,K=0=true,V=0=ARROW")
  )
),
createCyHiddenAttributes(
  name=c("A","B"),
  value=list(c("a1","a2"), "b")
),
createCyNetworkRelations(
  child = c(0,1),
  name = c("Network A", NA)
),
createCySubNetworks(
  nodes = list("all", c(0,1,2)),
  edges = list("all", c(0,1))
),
createCyTableColumn(
  appliesTo=c("nodes","edges","networks"),
  name=c("weight","weight","collapsed"),
  dataType=c("double","double","boolean")
)
```
rcxToJson

Description

Functions for converting the different aspects to JSON following the CX data structure definition (see NDEx documentation: https://home.ndexbio.org/data-model/).

Usage

rcxToJson(aspect, verbose = FALSE, ...)

## Default S3 method:
rcxToJson(aspect, verbose = FALSE, ...)

## S3 method for class 'MetaDataAspect'
rcxToJson(aspect, verbose = FALSE, ...)

## S3 method for class 'NodesAspect'
rcxToJson(aspect, verbose = FALSE, ...)

## S3 method for class 'EdgesAspect'
rcxToJson(aspect, verbose = FALSE, ...)

## S3 method for class 'NodeAttributesAspect'
rcxToJson(aspect, verbose = FALSE, ...)

## S3 method for class 'EdgeAttributesAspect'
rcxToJson(aspect, verbose = FALSE, ...)

## S3 method for class 'NetworkAttributesAspect'
rcxToJson(aspect, verbose = FALSE, ...)

## S3 method for class 'CartesianLayoutAspect'
rcxToJson(aspect, verbose = FALSE, ...)

## S3 method for class 'CyGroupsAspect'
rcxToJson(aspect, verbose = FALSE, ...)

## S3 method for class 'CyHiddenAttributesAspect'
rcxToJson(aspect, verbose = FALSE, ...)

## S3 method for class 'CyNetworkRelationsAspect'
rcxToJson(aspect, verbose = FALSE, ...)

## S3 method for class 'CySubNetworksAspect'
rcxToJson(aspect, verbose = FALSE, ...)

## S3 method for class 'CyTableColumnAspect'
rcxToJson(aspect, verbose = FALSE, ...)

## S3 method for class 'CyVisualPropertiesAspect'
rcxToJson(aspect, verbose = FALSE, ...)

## S3 method for class 'CyVisualProperty'
rcxToJson(aspect, verbose = FALSE, propertyOf = "", ...)

## S3 method for class 'CyVisualPropertyProperties'
rcxToJson(aspect, verbose = FALSE, ...)

## S3 method for class 'CyVisualPropertyDependencies'
rcxToJson(aspect, verbose = FALSE, ...)

## S3 method for class 'CyVisualPropertyMappings'
rcxToJson(aspect, verbose = FALSE, ...)

Arguments

- **aspect**: aspects of an RCX object
- **verbose**: logical; whether to print what is happening
- **...**: additional parameters, that might needed for extending
- **propertyOf**: character; provide propertyOf (only necessary for CyVisualProperty)

Details

For converting RCX objects to JSON, each aspect is processed by a generic function for its aspect class. Those functions return a character only containing the JSON of this aspect, which is then combined by toCX to be a valid CX data structure.

To support the conversion for non-standard or own-defined aspects, generic functions for those aspect classes have to be implemented.

Value

character; JSON of an aspect

See Also

toCX, writeCX, jsonToRCX, readCX

Examples

defines = createNodes(name = c("a","b","c","d","e","f"))
rcxToJson(nodes)
**readCX**

Read CX from file, parse the JSON and convert it to an RCX object

**Description**

The readCX function combines three sub-task:

- read the JSON from file
- parse the JSON
- process the contained aspects to create an RCX object

**Usage**

```r
readCX(file, verbose = FALSE, aspectClasses = getAspectClasses())
readJSON(file, verbose = FALSE)
parseJSON(json, verbose = FALSE)
processCX(aspectList, verbose = FALSE, aspectClasses = getAspectClasses())
```

**Arguments**

- `file` character; the name of the file which the data are to be read from
- `verbose` logical; whether to print what is happening
- `aspectClasses` named character; accession names and aspect classes
- `json` character; raw JSON data
- `aspectList` list; list containing the aspect data (parsed JSON)

**Details**

If any errors occur during this process, the single steps can be performed individually. This also allows to skip certain steps, for example if the JSON data is already available as text, there is no need to save it as file and read it again.

**Read the JSON from file:**

The readJSON function only read the content of a text file and returns it as a simple character vector.

**Parse the JSON:**

The parseJSON function uses the jsonlite package, to parse JSON text:

```r
jsonlite::fromJSON(cx, simplifyVector = FALSE)
```

The result is a list containing the aspect data as elements. If, for some reason, the JSON is not valid, the jsonlite package raises an error.
**Process the contained aspects to create an RCX object:**

With the `processCX` function, the single elements from the previous list will be processed with the `jsonToRCX` functions, which creating objects for the single aspects. The standard CX aspects are processed by generic functions named by the aspect names of the CX data structure, e.g. `jsonToRCX.nodeAttributes` for the same named CX aspect the corresponding `NodeAttributesAspect` in RCX (see also vignette("02. The RCX and CX Data Model") or NDEx documentation: https://home.ndexbio.org/data-model/).

The CX network may contain additional aspects besides the officially defined ones. This includes self defined or deprecated aspects, that still can be found in the networks at the NDEx platform. By default, those aspects are simply omitted. In those cases, the setting `verbose` to `TRUE` is a good idea to see, which aspects cannot be processed this package.

Those not processable aspects can be handled individually, but it is advisable to extend the `jsonToRCX` functions by implementing own versions for those aspects. Additionally, the `update` functions have to be implemented to add the newly generated aspect objects to RCX object (see e.g. `updateNodes` or `updateEdges`). Therefore, the function also have to be named "update<aspect-name>", where aspect-name is the capitalized version of the name used in the CX. (see also vignette("03. Extending the RCX Data Model")

**Value**

RCX object

**Functions**

- `readJSON`: Reads the CX/JSON from file and returns the content as text
- `parseJSON`: Parses the JSON text and returns a list with the aspect data
- `processCX`: Processes the list of aspect data and creates an RCX

**See Also**

`jsonToRCX`, `writeCX`

**Examples**

cxFile = system.file(  
  "extdata",  
  "Imatinib-Inhibition-of-BCR-ABL-66a902f5-2022-11e9-bb6a-0ac135e8bacf.cx",  
  package = "RCX"  
)

rcx = readCX(cxFile)

## OR:

json = readJSON(cxFile)
aspectList = parseJSON(json)
rcx = processCX(aspectList)
referredBy

List the aspects that are refered by an other aspect

Description

This function returns a list of all aspects with all present aspects, that refer to it. As example, the aspect NodesAspect is refered by the property source and target of the EdgesAspect aspect.

Usage

referredBy(rcx, aspectClasses = getAspectClasses())

Arguments

rcx an object of one of the aspect classes (e.g. NodesAspect, EdgesAspect, etc.)
aspectClasses named character; accession names and aspect classes aspectClasses

Value
	named list; Aspect class names with names of aspect classes, that refer to them.

Note

Uses hasIds() and refersTo() to determine the referring aspects.

See Also

hasIds(), idProperty(), refersTo(), maxId()

Examples

nodes = createNodes(name = c("CDK1","CDK2","CDK3"))
edges = createEdges(source = c(0,0), target = c(1,2))
rcx = createRCX(nodes = nodes, edges = edges)
referredBy(rcx)
**Description**

This function returns the name of the property and the aspect class it refers to. As example, the aspect `EdgesAspect` has the property `source` that refers to the `ids` of the `NodesAspect` aspect.

**Usage**

```r
refersTo(aspect)
## Default S3 method:
refersTo(aspect)
## S3 method for class 'EdgesAspect'
refersTo(aspect)
## S3 method for class 'NodeAttributesAspect'
refersTo(aspect)
## S3 method for class 'EdgeAttributesAspect'
refersTo(aspect)
## S3 method for class 'CartesianLayoutAspect'
refersTo(aspect)
## S3 method for class 'CyGroupsAspect'
refersTo(aspect)
## S3 method for class 'CyVisualPropertiesAspect'
refersTo(aspect)
## S3 method for class 'CySubNetworksAspect'
refersTo(aspect)
```

**Arguments**

- `aspect`: an object of one of the aspect classes (e.g. `NodesAspect`, `EdgesAspect`, etc.)

**Details**

Uses method dispatch, so the default return is `NULL` and only aspect classes that refer to other aspects are implemented. This way it is easier to extend the data model.

**Value**

- named list; Name of the referring property and aspect class name.
setExtension

Methods (by class)

- default: of default returns NULL
- EdgesAspect: of EdgesAspect refers to id by source and target
- NodeAttributesAspect: of NodeAttributesAspect refers to id by propertyOf and to id by subnetworkId
- EdgeAttributesAspect: of EdgeAttributesAspect refers to id by propertyOf and to id by subnetworkId
- CartesianLayoutAspect: of CartesianLayoutAspect refers to id by node and to id by view
- CyGroupsAspect: of CyGroupsAspect refers to id by nodes and to id by externalEdges and internalEdges
- CyVisualPropertiesAspect: of CyVisualPropertiesAspect refers to id by appliesTo of the sub-aspects
- CySubNetworksAspect: of refers to id by nodes and to id by edges

See Also

hasIds(), idProperty(), referredBy(), maxId()

Examples

edges = createEdges(source = c(0,0), target = c(1,2))
refersTo(edges)

Description

To simplify the usage of extension of the RCX data model new extensions can easily registered on load with this function. Registered extension then automatically are used for the conversion of CX data containing aspects of these extensions. The accession names and classes then are also added to getAspectClasses.

Usage

setExtension(package, accession, className)

Arguments

- package character; name of the extension package
- accession character; accession name used in RCX (e.g. rcx$accessionName)
- className character; class name of the aspect (e.g. is(rcx$accessionName, "AccessionNameAspect"))
Value

options()$RCX.options$extensions

See Also

aspectClasses

Examples

## Not run:
setExtension("RCXMyRcxExtension", "myRcxExtension", "MyRcxExtensionAspect")

## End(Not run)

summary
RCX and aspect summary

Description

summary is a generic function used to produce result summaries of the RCX object. The function invokes particular methods which depend on the class of the first argument.

Usage

## S3 method for class 'RCX'
summary(object, ...)

## S3 method for class 'MetaDataAspect'
summary(object, ...)

## S3 method for class 'NodesAspect'
summary(object, ...)

## S3 method for class 'EdgesAspect'
summary(object, ...)

## S3 method for class 'NodeAttributesAspect'
summary(object, ...)

## S3 method for class 'EdgeAttributesAspect'
summary(object, ...)

## S3 method for class 'NetworkAttributesAspect'
summary(object, ...)

## S3 method for class 'CartesianLayoutAspect'
summary(object, ...)
## S3 method for class 'CyGroupsAspect'
summary(object, ...)

## S3 method for class 'CyHiddenAttributesAspect'
summary(object, ...)

## S3 method for class 'CyNetworkRelationsAspect'
summary(object, ...)

## S3 method for class 'CySubNetworksAspect'
summary(object, ...)

## S3 method for class 'CyTableColumnAspect'
summary(object, ...)

## S3 method for class 'CyVisualPropertiesAspect'
summary(object, ...)

## S3 method for class 'CyVisualProperty'
summary(object, ...)

## S3 method for class 'AspectIdColumn'
summary(object, ...)

## S3 method for class 'AspectRefColumn'
summary(object, ...)

## S3 method for class 'AspectReqRefColumn'
summary(object, ...)

## S3 method for class 'AspectValueColumn'
summary(object, ...)

## S3 method for class ' AspectAttributeColumn'
summary(object, ...)

## S3 method for class ' AspectListLengthColumn'
summary(object, ...)

**Arguments**

object 

an object: RCX object or aspect (or column of data.frame)

... 

additional arguments affecting the summary produced.

**Details**

The form of the returned summary depends on the class of its argument, therefore it is possible to summarize RCX objects and their single aspects.
To enhance readability of the summary, some additional classes have `summary` functions, that are used to show for example ids of an aspect, required and optional references to ids of aspects, or the number of elements in lists.

**Value**

object summary as list

**Methods (by class)**

- `AspectIdColumn`: Summarize an id property
- `AspectRefColumn`: Summarize an optional property, that references the ids of an other aspect
- `AspectReqRefColumn`: Summarize a required property, that references the ids of an other aspect
- `AspectValueColumn`: Summarize the occurrences of the different elements in the property
- `AspectAttributeColumn`: Summarize the different attributes in the property
- `AspectListLengthColumn`: The property is a list of vectors, so summarize the length of the vectors

**Examples**

```r
rcx = createRCX(
  nodes = createNodes(name = c("a","b","c")),
  edges = createEdges(source=1, target=2)
)
summary(rcx)
```

---

**toCX**  
*Convert an RCX object to CX (JSON)*

**Description**

This function converts an RCX object to JSON in a valid CX data structure (see NDEx documentation: [https://home.ndexbio.org/data-model/](https://home.ndexbio.org/data-model/)).

**Usage**

`toCX(rcx, verbose = FALSE, pretty = FALSE)`

**Arguments**

- `rcx`  
  RCX object
- `verbose`  
  logical; whether to print what is happening
- `pretty`  
  logical; adds indentation whitespace to JSON output. Can be TRUE/FALSE or a number specifying the number of spaces to indent. See `jsonlite::prettify()`
Details

The single aspects of the RCX object are processed by generic functions of \texttt{rcxToJson} for each aspect class. Therefore, not only the single aspects are converted to JSON, but also necessary additional aspects are added, so the resulting CX is accepted by the NDEx platform (\url{https://ndexbio.org/}):

- \textit{numberVerication} shows the supported maximal number
- \textit{status} is needed at the end to show, that no errors have occurred while creation

If the RCX object contains additional aspects besides the officially defined ones, the corresponding \texttt{rcxToJson} functions for those aspect classes have to be implemented in order to include them in the resulting CX.

Value

CX (JSON) text

See Also

toCX, rcxToJson, readCX, writeCX

Examples

```r
rcx = createRCX(
    nodes = createNodes(
        name = LETTERS[seq_len(10)]
    ),
    edges = createEdges(
        source = c(1,2),
        target = c(2,3)
    )
)

json = toCX(rcx, pretty=TRUE)
```

---

**updateCartesianLayout**  
*Update Cartesian Layouts*

**Description**

This function adds a cartesian layout in the form of a \texttt{CartesianLayout} object to an other \texttt{CartesianLayout} or an RCX object.
updateCartesianLayout

Usage

updateCartesianLayout(
  x,
  cartesianLayout,
  replace = TRUE,
  stopOnDuplicates = FALSE,
  ...
)

## S3 method for class 'CartesianLayoutAspect'
updateCartesianLayout(
  x,
  cartesianLayout,
  replace = TRUE,
  stopOnDuplicates = FALSE,
  ...
)

## S3 method for class 'RCX'
updateCartesianLayout(
  x,
  cartesianLayout,
  replace = TRUE,
  stopOnDuplicates = FALSE,
  checkReferences = TRUE,
  ...
)

Arguments

x

RCX or CartesianLayout object; (to which the new layout will be added)
cartesianLayout

CartesianLayout object; (the layout, that will be added)
replace

logical; if existing values are updated (or ignored)
stopOnDuplicates

logical; whether to stop, if duplicates in nodes (and view if present) column are found

... additional parameters
checkReferences

logical; whether to check if references to other aspects are present in the RCX object

Details

Networks, or more precisely its nodes may have a cartesian layout, that is represented as CartesianLayout object. CartesianLayout objects can be added to an RCX or an other CartesianLayout object.
In the case, that a CartesianLayout object is added to an other, or the RCX object already contains a CartesianLayout object, some attributes might be present in both. By default, the properties are updated with the values of the latest one. This can prevented by setting the replace parameter to FALSE, in that case only new properties are added and the existing properties remain untouched.

Furthermore, if duplicated properties are considered as a preventable mistake, an error can be raised by setting stopOnDuplicates to TRUE. This forces the function to stop and raise an error, if duplicated properties are present.

Value

CartesianLayoutAspect or RCX object with added layout

Examples

```r
## For CartesianLayoutAspects:
## prepare some aspects:
cartesianLayout = createCartesianLayout(
  node=c(0, 1),
  x=c(5.5, 110.1),
  y=c(200.3, 210.2),
  z=c(-1, 3.1),
)

## node 0 is updated, new view is added
cartesianLayout2 = createCartesianLayout(
  node=c(0, 0),
  x=c(5.7, 7.2),
  y=c(98, 13.9),
  view=c(NA, 1476)
)

## Simply update with new values
cartesianLayout3 = updateCartesianLayout(cartesianLayout, cartesianLayout2)

## Ignore already present keys
cartesianLayout3 = updateCartesianLayout(cartesianLayout, cartesianLayout2, replace=FALSE)

## Raise an error if duplicate keys are present
try(updateCartesianLayout(cartesianLayout, cartesianLayout2, stopOnDuplicates=TRUE))

## =>ERROR:
## Provided IDs (node, view) countain duplicates!

## For RCX:
## prepare RCX object:
## nodes = createNodes(name = c("a","b"))
## edges = createEdges(source = 0, target = 1)
cySubNetworks = createCySubNetworks(
  id = 1476,
  nodes = "all",
  edges = "all"
)```
rcx = createRCX(nodes,
    edges = edges,
    cySubNetworks=cySubNetworks)

## add the network attributes
rcx = updateCartesianLayout(rcx, cartesianLayout)

## add additional network attributes and update existing
rcx = updateCartesianLayout(rcx, cartesianLayout2)

updateCyGroups  
Update Cytoscape Groups

Description
This functions add Cytoscape groups in the form of a CyGroups object to an RCX or an other CyGroups object.

Usage
updateCyGroups(x, cyGroups, stopOnDuplicates = FALSE, keepOldIds = TRUE, ...)

## S3 method for class 'CyGroupsAspect'
updateCyGroups(x, cyGroups, stopOnDuplicates = FALSE, keepOldIds = TRUE, ...)

## S3 method for class 'RCX'
updateCyGroups(
    x,
    cyGroups,
    stopOnDuplicates = FALSE,
    keepOldIds = TRUE,
    checkReferences = TRUE,
    ...
)

Arguments

x  
   RCX or CyGroups object; (to which the new Cytoscape groups will be added)

cyGroups  
   CyGroups object; (the new aspect, that will be added)

stopOnDuplicates  
   logical; whether to stop, if duplicates in id column are found, or re-assign ids instead.

keepOldIds  
   logical; if ids are re-assigned, the original ids are kept in the column oldId

...  
   additional parameters

checkReferences  
   logical; whether to check if references to other aspects are present in the RCX object
updateCyGroups

Details

Cytoscape groups allow to group a set of nodes and corresponding internal and external edges together, and represent a group as a single node in the visualization. CyGroups objects can be added to an RCX or another CyGroups object. The nodes, internalEdges and externalEdges parameters reference the node or edge IDs that belong to a group. When adding an CyGroups object to an RCX object, those IDs must be present in the Nodes or Edges aspect respectively, otherwise an error is raised.

When two groups should be added to each other some conflicts may rise, since the aspects might use the same IDs. If the aspects do not share any IDs, the two aspects are simply combined. Otherwise, the IDs of the new groups are re-assigned continuing with the next available ID (i.e. \( \text{maxId} (\text{cyGroupsAspect}) + 1 \) and \( \text{maxId} (\text{rcx$cyGroups}) + 1 \), respectively).

To keep track of the changes, it is possible to keep the old IDs of the newly added nodes in the automatically added column oldId. This can be omitted by setting keepOldIds to FALSE. Otherwise, if a re-assignment of the IDs is not desired, this can be prevented by setting stopOnDuplicates to TRUE. This forces the function to stop and raise an error, if duplicated IDs are present.

Value

CyGroups or RCX object with added Cytoscape groups

See Also

CyGroups;

Examples

```r
## For CyGroupsAspects:
## prepare some aspects:
cyGroups1 = createCyGroups(
  name = c("Group One", "Group Two"),
  nodes = list(c(1,2,3), 0),
  internalEdges = list(c(0,1),NA),
  externalEdges = list(NA,c(2,3)),
  collapsed = c(TRUE,NA)
)
cyGroups2 = createCyGroups(
  name = "Group Three",
  nodes = list(c(4,5)),
  externalEdges = list(c(4,5))
)

## group ids will be kept
cyGroups3 = updateCyGroups(cyGroups1, cyGroups2)

## old group ids will be omitted
keepOldIds=FALSE)
cyGroups3 = updateCyGroups(cyGroups1, cyGroups2, keepOldIds=FALSE)

## Raise an error if duplicate keys are present
```
try(updateCyGroups(cyGroups1, cyGroups2,
         stopOnDuplicates=TRUE))
## =>ERROR:
## Elements of "id" (in updateCyGroups) must not contain duplicates!

## For RCX
## prepare RCX object:
nodes = createNodes(name = c("a","b","c","d","e","f"))
edges = createEdges(source=c(1,2,0,0,0,2),
                    target=c(2,3,1,2,5,4))
rcx = createRCX(nodes, edges)

## add the group
rcx = updateCyGroups(rcx, cyGroups1)

## add an additional group
rcx = updateCyGroups(rcx, cyGroups2)

## create a group with a not existing node...
cyGroups3 = createCyGroups(
    name = "Group Three",
    nodes = list(9)
)
## ...and try to add them
try(updateCyGroups(rcx, cyGroups3))
## =>ERROR:
## Provided IDs of "additionalGroups$nodes" (in updateCyGroups)
## don't exist in "rcx$nodes$id"

## create a group with a not existing edge...
cyGroups4 = createCyGroups(
    name = "Group Four",
    nodes = list(c(1,2)),
    internalEdges = list(c(9))
)
## ...and try to add them
try(updateCyGroups(rcx, cyGroups4))
## =>ERROR:
## Provided IDs of "additionalGroups$internalEdges" (in updateCyGroups)
## don't exist in "rcx$edges$id"

updateCyHiddenAttributes

Update Cytoscape hidden attributes

Description

This function adds hidden attributes in the form of a CyHiddenAttributes object to another CyHiddenAttributes or an RCX object.
updateCyHiddenAttributes

Usage
updateCyHiddenAttributes(
  x,
  hiddenAttributes,
  replace = TRUE,
  stopOnDuplicates = FALSE,
  ...
)

## S3 method for class 'CyHiddenAttributesAspect'
updateCyHiddenAttributes(
  x,
  hiddenAttributes,
  replace = TRUE,
  stopOnDuplicates = FALSE,
  ...
)

## S3 method for class 'RCX'
updateCyHiddenAttributes(
  x,
  hiddenAttributes,
  replace = TRUE,
  stopOnDuplicates = FALSE,
  checkReferences = TRUE,
  ...
)

Arguments
x 
RCX or CyHiddenAttributes object; (to which the new hidden attributes will be added)

hiddenAttributes
CyHiddenAttributes object; (the new aspect, that will be added)

replace
logical; if existing values are updated (or ignored)

stopOnDuplicates
logical; whether to stop, if duplicates in name (and subnetworkId if present) column are found

... additional parameters

checkReferences
logical; whether to check if references to other aspects are present in the RCX object

Details
Cytoscape subnetworks allow to group a set of nodes and corresponding edges together, and network relations define the relations between those networks. CyHiddenAttributes objects can be added to an RCX or an other CyHiddenAttributes object.
In the case, that a CyHiddenAttributes object is added to an other, or the RCX object already contains a CyHiddenAttributes object, some attributes might be present in both. By default, the attributes are updated with the values of the latest one. This can prevented by setting the replace parameter to FALSE, in that case only new attributes are added and the existing attributes remain untouched.

Furthermore, if duplicated attributes are considered as a preventable mistake, an error can be raised by setting stopOnDuplicates to TRUE. This forces the function to stop and raise an error, if duplicated attributes are present.

Value

CyHiddenAttributes or RCX object with added hidden attributes

Examples

```r
## For CyHiddenAttributesAspects:
## prepare some aspects:
hiddenAttributes1 = createCyHiddenAttributes(
  name=c("A","A","B","B"),
  value=list(c("a1","a2"),
             "a with subnetwork",
             "b",
             "b with subnetwork"),
  isList=c(TRUE,FALSE,TRUE,FALSE),
  subnetworkId=c(NA,1,NA,1))

## A is updated, C is new
hiddenAttributes2 = createCyHiddenAttributes(
  name=c("A","A","C"),
  value=list("new a",
             "new a with subnetwork",
             c(1,2)),
  subnetworkId=c(NA,1,NA))

## Simply update with new values
hiddenAttributes3 = updateCyHiddenAttributes(hiddenAttributes1, hiddenAttributes2)

## Ignore already present keys
hiddenAttributes3 = updateCyHiddenAttributes(hiddenAttributes1, hiddenAttributes2, replace=FALSE)

## Raise an error if duplicate keys are present
try(updateCyHiddenAttributes(hiddenAttributes1, hiddenAttributes2, stopOnDuplicates=TRUE))
## =>ERROR:
## Elements of "name" and "subnetworkId" (in updateCyHiddenAttributes) must not contain duplicates!
```
## For RCX
### prepare RCX object:
```r
nodes = createNodes(name = c("a","b","c","d","e","f"))
edges = createEdges(source=c(1,2,0,0,0,2),
                   target=c(2,3,1,2,5,4))
rcx = createRCX(nodes, edges)
cySubNetworks = createCySubNetworks(
  id = c(1,2),
  nodes = list("all", c(1,2,3)),
  edges = list("all", c(0,2))
)
rcx = updateCySubNetworks(rcx, cySubNetworks)
```

### add a network relation
```r
rcx = updateCyHiddenAttributes(rcx, hiddenAttributes1)
```

### add an additional relation (update with new values)
```r
rcx = updateCyHiddenAttributes(rcx, hiddenAttributes2)
```

### create a relation with a not existing subnetwork...
```r
hiddenAttributes3 = createCyHiddenAttributes(
  name="X",
  value="new x",
  subnetworkId=9
)
```

### ...and try to add them
```r
try(updateCyHiddenAttributes(rcx, hiddenAttributes3))
```

### =>ERROR:
```r
# Provided IDs of "additionalAttributes$subnetworkId" (in updateCyHiddenAttributes)
# don't exist in "rcx$cySubNetworks$id"
```

---

**Description**

This functions add network relations in the form of a `CyNetworkRelations` object to an other `CyNetworkRelations` or an RCX object.

**Usage**

```r
updateCyNetworkRelations(
  x,
  cyNetworkRelations,
  replace = TRUE,
  stopOnDuplicates = FALSE,
  ...
)```
updateCyNetworkRelations

## S3 method for class 'CyNetworkRelationsAspect'
updateCyNetworkRelations(
  x,
  cyNetworkRelations,
  replace = TRUE,
  stopOnDuplicates = FALSE,
  ...
)

## S3 method for class 'RCX'
updateCyNetworkRelations(
  x,
  cyNetworkRelations,
  replace = TRUE,
  stopOnDuplicates = FALSE,
  checkReferences = TRUE,
  ...
)

### Arguments

- **x**
  - RCX or CySubNetworks object; (to which the new network relations will be added)
- **cyNetworkRelations**
  - CySubNetworks object; (the network relations, that will be added)
- **replace**
  - logical; if existing values are updated (or ignored)
- **stopOnDuplicates**
  - logical; whether to stop, if duplicates in the child column are found
- **checkReferences**
  - logical; whether to check if references to other aspects are present in the RCX object

### Details

Cytoscape subnetworks allow to group a set of nodes and corresponding edges together, and network relations define the relations between those networks. CyNetworkRelations objects can be added to an RCX or an other CyNetworkRelations object.

When network relations are added to a CyNetworkRelations or a RCX object some conflicts may rise, since the aspects might use the same child IDs. If the aspects do not share any child IDs, the two aspects are simply combined, otherwise, the properties of the child are updated. If that is not wanted, the updating can be prevented by setting replace to FALSE. Furthermore, if duplicated properties are considered as a preventable mistake, an error can be raised by setting stopOnDuplicates to TRUE. This forces the function to stop and raise an error, if duplicated child IDs are present.
Value

CyNetworkRelations or RCX object with added network relations

Examples

## For CyNetworkRelationsAspects:
## prepare some aspects:
cyNetworkRelations1 = createCyNetworkRelations(
    child = c(1,2),
    parent = c(NA,1),
    name = c("Network A",
            "View A"),
    isView = c(FALSE, TRUE)
)
cyNetworkRelations2 = createCyNetworkRelations(
    child = 2,
    name = "View B",
    isView = TRUE
)
## update the duplicated child
cyNetworkRelations3 = updateCyNetworkRelations(cyNetworkRelations1,
                                            cyNetworkRelations2)
## keep old child values
cyNetworkRelations4 = updateCyNetworkRelations(cyNetworkRelations1,
                                            cyNetworkRelations2,
                                            replace=FALSE)

## Raise an error if duplicate keys are present
try(updateCyNetworkRelations(cyNetworkRelations1,
                              cyNetworkRelations2,
                              stopOnDuplicates=TRUE))
## =>ERROR:
## Elements of "child" (in updateCyNetworkRelations)
## must not contain duplicates!

## For RCX
## prepare RCX object:
nodes = createNodes(name = c("a","b","c","d","e","f"))
edges = createEdges(source=c(1,2,0,0,0,2),
                    target=c(2,3,1,2,5,4))
rcx = createRCX(nodes, edges)
cySubNetworks = createCySubNetworks(
    id = c(1,2),
    nodes = list("all", c(1,2,3)),
    edges = list("all", c(0,2))
)
rcx = updateCySubNetworks(rcx, cySubNetworks)
## add a network relation
rcx = updateCyNetworkRelations(rcx, cyNetworkRelations1)

## add an additional relation (View A is replaced by B)
rcx = updateCyNetworkRelations(rcx, cyNetworkRelations2)

## create a relation with a not existing subnetwork...
cyNetworkRelations3 = createCyNetworkRelations(
  child = 9
)

## ...and try to add them
try(updateCyNetworkRelations(rcx, cyNetworkRelations3))
## =>ERROR:
## Provided IDs of "additionalNetworkRelations$child" (in addCyNetworkRelations)
## don't exist in "rcx$cySubNetworks$id"

## create a relation with a not existing parent subnetwork...
cyNetworkRelations4 = createCyNetworkRelations(
  child = 1,
  parent = 9
)

## ...and try to add them
try(updateCyNetworkRelations(rcx, cyNetworkRelations4))
## =>ERROR:
## Provided IDs of "additionalNetworkRelations$parent" (in addCyNetworkRelations)
## don't exist in "rcx$cySubNetworks$id"

updateCySubNetworks  Update Cytoscape subnetworks

**Description**

This functions add subnetworks in the form of a `CySubNetworks` object to an other `CySubNetworks` or an `RCX` object.

**Usage**

```r
updateCySubNetworks(
  x,
  cySubNetworks,
  stopOnDuplicates = FALSE,
  keepOldIds = TRUE,
  ...
)
```

## S3 method for class 'CySubNetworksAspect'
updateCySubNetworks(
  x,
  ...)
Arguments

- **x**
  - RCX or CySubNetworks object; (to which the new subnetworks will be added)

- **cySubNetworks**
  - CySubNetworks object; (the subnetwork, that will be added)

- **stopOnDuplicates**
  - logical; whether to stop, if duplicates in id column are found, or re-assign ids instead.

- **keepOldIds**
  - logical; if ids are re-assigned, the original ids are kept in the column oldId

- **...**
  - additional parameters

- **checkReferences**
  - logical; whether to check if references to other aspects are present in the RCX object

Details

Cytoscape subnetworks allow to group a set of nodes and corresponding edges together. CySubNetworks objects can be added to an RCX or an other CySubNetworks object. The nodes and edges parameters reference the node or edge IDs that belong to a subnetwork. When adding an CySubNetworks object to an RCX object, those IDs must be present in the Nodes or Edges aspect respectively, otherwise an error is raised. Unlike other aspects referring those IDs, the Cytoscape subnetwork aspect allows to refer to all nodes and edges using the keyword all.

When subnetworks should be added to a CySubNetworks or a RCX object some conflicts may rise, since the aspects might use the same IDs. If the aspects do not share any IDs, the two aspects are simply combined. Otherwise, the IDs of the new subnetworks are re-assinged continuing with the next available ID (i.e. maxId(cySubNetworks) + 1 and maxId(rcx$cySubNetworks) + 1, respectively).

To keep track of the changes, it is possible to keep the old IDs of the newly added nodes in the automatically added column oldId. This can be omitted by setting keepOldIds to FALSE. Otherwise, if a re-assignment of the IDs is not desired, this can be prevented by setting stopOnDuplicates to TRUE. This forces the function to stop and raise an error, if duplicated IDs are present.
updateCySubNetworks

Value

CySubNetworks or RCX object with added subnetworks

See Also

CyNetworkRelations;

Examples

## For CySubNetworksAspects:
## prepare some aspects:
cySubNetworks1 = createCySubNetworks(
id = c(0,1),
nodes = list("all",
c(1,2,3)),
edges = list("all",
c(0,2))
)
cySubNetworks2 = createCySubNetworks(
nodes = c(0,3),
edges = c(1)
)

## subnetwork ids will be kept
cySubNetworks3 = updateCySubNetworks(cySubNetworks1, cySubNetworks2)

## old subnetwork ids will be omitted
cySubNetworks3 = updateCySubNetworks(cySubNetworks1, cySubNetworks2, keepOldIds=FALSE)

## Raise an error if duplicate keys are present
try(updateCySubNetworks(cySubNetworks1, cySubNetworks2, stopOnDuplicates=TRUE))
## =>ERROR:
## Elements of "id" (in updateCySubNetworks) must not contain duplicates!

## For RCX
## prepare RCX object:
nodes = createNodes(name = c("a","b","c","d","e","f"))
edges = createEdges(source=c(1,2,0,0,0,2),
target=c(2,3,1,2,5,4))
rcx = createRCX(nodes, edges)

## add the subnetwork
rcx = updateCySubNetworks(rcx, cySubNetworks1)

## add additional subnetwork
rcx = updateCySubNetworks(rcx, cySubNetworks2)

## create a subnetwork with a not existing node...
cySubNetworks3 = createCySubNetworks(
## Description

This function adds hidden attributes in the form of a `CyTableColumn` object to another `CyTableColumn` or an `RCX` object.

## Usage

```r
updateCyTableColumn(
  x,
  cyTableColumns,
  replace = TRUE,
  stopOnDuplicates = FALSE,
  ...
)
```

```r
## S3 method for class 'CyTableColumnAspect'
updateCyTableColumn(
  x,
  cyTableColumns,
  replace = TRUE,
  stopOnDuplicates = FALSE,
  ...
)
```

```r
## S3 method for class 'RCX'
updateCyTableColumn(
  x,
  cyTableColumns,
  replace = TRUE,
  stopOnDuplicates = FALSE,
  ...
)
```

## Examples

```r
nodes = list(9)

## ...and try to add them
try(updateCySubNetworks(rcx, cySubNetworks3))
## =>ERROR:
## Provided IDs of "additionalSubNetworks$nodes" (in addCySubNetworks)
## don't exist in "rcx$nodes$id"

## create a group with a not existing edge...
cySubNetworks4 = createCySubNetworks(
  nodes = c(0,1),
  edges = 9
)

## ...and try to add them
try(updateCySubNetworks(rcx, cySubNetworks4))
## =>ERROR:
## Provided IDs of "additionalSubNetworks$edges" (in addCySubNetworks)
## don't exist in "rcx$edges$id"
```
updateCyTableColumn(x, 
  cyTableColumns, 
  replace = TRUE, 
  stopOnDuplicates = FALSE, 
  checkReferences = TRUE, 
  ...
)

Arguments

x            RCX or CyTableColumn object; (to which the new table column properties will be added)
cyTableColumns CyTableColumn object; (the new aspect, that will be added)
replace       logical; if existing values are updated (or ignored)
stopOnDuplicates logical; whether to stop, if duplicates in appliesTo and name’ (and subnetworkId if present) column are found
...           additional parameters
checkReferences logical: whether to check if references to other aspects are present in the RCX object

Details

In the case, that a CyTableColumn object is added to an other, or the RCX object already contains a CyTableColumn object, some properties might be present in both. By default, the properties are updated with the values of the latest one. This can prevented by setting the replace parameter to FALSE, in that case only new attributes are added and the existing attributes remain untouched.

Furthermore, if duplicated properties are considered as a preventable mistake, an error can be raised by setting stopOnDuplicates to TRUE. This forces the function to stop and raise an error, if duplicated properties are present.

Cytoscape does not currently support table columns for the root network, but this is option is included here for consistency.

Value

CyTableColumn or RCX object with added hidden attributes

See Also

CySubNetworks

Examples

## For CyTableColumnssAspects:
## prepare some aspects:
tableColumn1 = createCyTableColumn(
applyTo=c("nodes","edges","networks"),
name=c("weight","weight","collapsed"),
dataType=c("numeric","double","logical"),
isList=c(FALSE,FALSE,TRUE),
subnetworkId=c(NA,NA,1))

## nodes is updated, networks is new
tableColumn2 = createCyTableColumn(
    applyTo=c("nodes","networks"),
    name=c("weight","collapsed"),
    dataType=c("double","character")
)

## Simply update with new values
tableColumn3 = updateCyTableColumn(tableColumn1, tableColumn2)

## Ignore already present keys
tableColumn3 = updateCyTableColumn(tableColumn1, tableColumn2,
    replace=FALSE)

## Raise an error if duplicate keys are present
try(updateCyTableColumn(tableColumn1, tableColumn2,
    stopOnDuplicates=TRUE))

## =>ERROR:
## Elements of "appliesTo", "name" and "subnetworkId" (in updateCyTableColumn)
## must not contain duplicates!

## For RCX
## prepare RCX object:
nodes = createNodes(name = c("a","b","c","d","e","f"))
edges = createEdges(source=c(1,2,0,0,0,2),
    target=c(2,3,1,2,5,4))
rcx = createRCX(nodes, edges)
cySubNetworks = createCySubNetworks(
    id = c(1,2),
    nodes = list("all", c(1,2,3)),
    edges = list("all", c(0,2))
)
rcx = updateCySubNetworks(rcx, cySubNetworks)

## add a table column property
rcx = updateCyTableColumn(rcx, tableColumn1)

## add an additional property (update with new values)
rctx = updateCyTableColumn(rcx, tableColumn2)

## create a property with a not existing subnetwork...
tableColumn3 = createCyTableColumn(
    applyTo="nodes",
    name="weight",
    subnetworkId=9
)
## updateCyVisualProperties

### Description

This function is used to add Cytoscape visual properties aspects to each other or to an RCX object. In a CyVisualProperties aspect, CyVisualProperty objects define networks, nodes, edges, and default nodes and edges.

### Usage

```r
updateCyVisualProperties(
  x,
  cyVisualProperties,
  replace = TRUE,
  stopOnDuplicates = FALSE,
  ...
)
```

```r
## S3 method for class 'CyVisualPropertiesAspect'
updateCyVisualProperties(
  x,
  cyVisualProperties,
  replace = TRUE,
  stopOnDuplicates = FALSE,
  ...
)
```

```r
## S3 method for class 'RCX'
updateCyVisualProperties(
  x,
  cyVisualProperties,
  replace = TRUE,
  stopOnDuplicates = FALSE,
  checkReferences = TRUE,
  ...
)
```
Arguments

- `x`  
  RCX or `CyVisualProperties` object; (to which it will be added)
  
- `cyVisualProperties`  
  `CyVisualProperties` object; (that will be added)
  
- `replace`  
  logical; if existing values are updated (or ignored)
  
- `stopOnDuplicates`  
  logical; whether to stop, if duplicates in name (and subnetworkId if present) column are found
  
- ...  
  additional parameters
  
- `checkReferences`  
  logical; whether to check if references to other aspects are present in the RCX object

Details

**Structure of Cytoscape Visual Property:**

```
CyVisualProperty
|---properties = CyVisualPropertyProperties
  |  |---name
  |  |---value
|---dependencies = CyVisualPropertyDependencies
  |  |---name
  |  |---value
|---mappings = CyVisualPropertyMappings
  |  |---name
  |  |---type
  |  |---definition
|---appliesTo = <reference to subnetwork id>
|---view = <reference to subnetwork id>
```

`CyVisualProperties` aspects consist of `CyVisualProperty` objects for each entry: networks, nodes, edges, and default nodes and edges. Two `CyVisualProperties` aspects are merged by adding its entries individually.

`CyVisualProperty` objects differ in the sub-networks and views (`CySubNetworks`) they apply to, subsequently properties, dependencies and mappings are merged based on the uniqueness in those two.

Properties, dependencies and mappings (i.e. `CyVisualPropertyProperties`, `CyVisualPropertyDependencies` and `CyVisualPropertyMappings` objects) are unique in name. By default, the duplicate attributes are updated with the values of the latest one. This can prevented by setting the `replace` parameter to `FALSE`, in that case only new attributes are added and the existing attributes remain untouched. Furthermore, if duplicated attributes are considered as a preventable mistake, an error can be raised by setting `stopOnDuplicates` to `TRUE`. This forces the function to stop and raise an error, if duplicated attributes are present.

Value

`CyVisualProperties` or RCX object with added Cytoscape visual properties
See Also

updateCyVisualProperty, getCyVisualProperty

Examples

## Prepare used properties

## Visual property: Properties

vpPropertyP1 = createCyVisualPropertyProperties(c(NODE_BORDER_STROKE="SOLID"))
vpPropertyP2 = createCyVisualPropertyProperties(c(NODE_BORDER_WIDTH="1.5"))
vpPropertyP3 = createCyVisualPropertyProperties(c(NODE_BORDER_WIDTH="999"))

## Visual property: Dependencies

vpPropertyD1 = createCyVisualPropertyDependencies(c(nodeSizeLocked="false"))
vpPropertyD2 = createCyVisualPropertyDependencies(c(arrowColorMatchesEdge="true"))
vpPropertyD3 = createCyVisualPropertyDependencies(c(arrowColorMatchesEdge="false"))

## Visual property: Mappings

vpPropertyM1 = createCyVisualPropertyMappings(c(NODE_FILL_COLOR="CONTINUOUS"),
                                               "COL=directed,T=boolean,K=0=true,V=0=ARROW")
vpPropertyM2 = createCyVisualPropertyMappings(c(EDGE_TARGET_ARROW_SHAPE="DISCRETE"),
                                               "TRIANGLE")
vpPropertyM3 = createCyVisualPropertyMappings(c(EDGE_TARGET_ARROW_SHAPE="DISCRETE"),
                                               "NONE")

## Create visual property object

vpProperty1 = createCyVisualProperty(properties=list(vpPropertyP1,
                                                      vpPropertyP1),
                                       dependencies=list(vpPropertyD1,
                                                        NA),
                                       mappings=list(vpPropertyM1,
                                                        NA),
                                       appliesTo = c(NA,
                                                     1),
                                       view = c(NA,
                                                1))

vpProperty2 = createCyVisualProperty(properties=vpPropertyP2,
                                       dependencies=vpPropertyD2,
                                       mappings=vpPropertyM2)

vpProperty3 = createCyVisualProperty(properties=vpPropertyP3,
                                       dependencies=vpPropertyD3,
                                       mappings=vpPropertyM3)

## Create a visual properties aspect

## (using the same visual property object for simplicity)

visProp1 = createCyVisualProperties(network=vpProperty1,
                                     nodes=vpProperty1,
                                     edges=vpProperty1,
                                     defaultNodes=vpProperty1,
                                     defaultEdges=vpProperty1)

visProp2 = createCyVisualProperties(network=vpProperty2,
                                     nodes=vpProperty2,
                                     edges=vpProperty2,
updateCyVisualProperties

visProp3 = createCyVisualProperties(network=vpProperty3,
  nodes=vpProperty3,
  edges=vpProperty3,
  defaultNodes=vpProperty3,
  defaultEdges=vpProperty3)

## Adding a different visual property (Properties, Dependencies, Mappings)
## (e.g. "NODE_BORDER_WIDTH", which is not present before)
visProp4 = updateCyVisualProperties(visProp1, visProp2)

## Update a existing visual property
visProp5 = updateCyVisualProperties(visProp4, visProp3)

## Raise an error if duplicate keys are present
try(updateCyVisualProperties(visProp4, visProp3,
  stopOnDuplicates=TRUE))
## =>ERROR:
## Elements of name (in VisualProperties$network$properties<appliesTo=NA,view=NA>)
## must not contain duplicates!

## For RCX
## prepare RCX object:
nodes = createNodes(name = c("a","b","c","d","e","f"))
edges = createEdges(source=c(1,2,0,0,0,2),
  target=c(2,3,1,2,5,4))
rcx = createRCX(nodes, edges)
cySubNetworks = createCySubNetworks(
  id = c(1,2),
  nodes = list("all", c(1,2,3)),
  edges = list("all", c(0,2)))
rcx = updateCySubNetworks(rcx, cySubNetworks)

## Adding visual properties to an RCX object
rcx = updateCyVisualProperties(rcx, visProp1)

## Adding a different visual property (Properties, Dependencies, Mappings)
## (e.g. "NODE_BORDER_WIDTH", which is not present before)
rcx = updateCyVisualProperties(rcx, visProp2)

## Update a existing visual property
rcx = updateCyVisualProperties(rcx, visProp3)

## Raise an error if duplicate keys are present
try(updateCyVisualProperties(rcx, visProp3,
  stopOnDuplicates=TRUE))
## =>ERROR:
## Elements of "name" (in VisualProperties$network$properties<appliesTo=NA,view=NA>)
## must not contain duplicates!
updateCyVisualProperty

Update Cytoscape Visual Property objects and sub-objects (used in CyVisualProperties aspect)

Description

This function is used to add Cytoscape visual property objects (CyVisualProperty) and its sub-

Usage

updateCyVisualProperty(
  cyVisualProperty,
  additionalProperty,
  replace = TRUE,
  stopOnDuplicates = FALSE,
  .log = c()
)

## S3 method for class 'CyVisualPropertyProperties'
updateCyVisualProperty(
  cyVisualProperty,
  additionalProperty,
  replace = TRUE,
  stopOnDuplicates = FALSE,
  .log = c()
)

## S3 method for class 'CyVisualPropertyDependencies'
updateCyVisualProperty(
  cyVisualProperty,
  additionalProperty,
  replace = TRUE,
  stopOnDuplicates = FALSE,
  .log = c()
)

## S3 method for class 'CyVisualPropertyMappings'
updateCyVisualProperty(
  cyVisualProperty,
  additionalProperty,
  replace = TRUE,
  stopOnDuplicates = FALSE,
  .log = c()
## S3 method for class 'CyVisualProperty'
updateCyVisualProperty(
  cyVisualProperty,
  additionalProperty,
  replace = TRUE,
  stopOnDuplicates = FALSE,
  .log = c()
)

### Arguments

- **cyVisualProperty**: object; (to which it will be added)
- **additionalProperty**: object; (that will be added)
- **replace**: logical; if existing values are updated (or ignored)
- **stopOnDuplicates**: logical; whether to stop, if duplicates in name (and subnetworkId if present) column are found
- **.log**: character (optional); name of the calling function used in error logging

### Details

**Structure of Cytoscape Visual Property:**

- `CyVisualProperty`
  - `properties = CyVisualPropertyProperties`
  - `dependencies = CyVisualPropertyDependencies`
  - `mappings = CyVisualPropertyMappings`

`CyVisualProperty` objects differ in the sub-networks and views (`CySubNetworks`) they apply to, subsequently properties, dependencies and mappings are merged based on the uniqueness in those two.

Properties, dependencies and mappings (i.e. `CyVisualPropertyProperties`, `CyVisualPropertyDependencies` and `CyVisualPropertyMappings` objects) are unique in name. By default, the duplicate attributes are updated with the values of the latest one. This can prevented by setting the `replace` parameter to `FALSE`, in that case only new attributes are added and the existing attributes remain untouched. Furthermore, if duplicated attributes are considered as a preventable mistake, an error can be raised by
setting `stopOnDuplicates` to `TRUE`. This forces the function to stop and raise an error, if duplicated attributes are present.

Value

`CyVisualProperty, CyVisualPropertyProperties, CyVisualPropertyDependencies or CyVisualPropertyMappings` objects

See Also

`getCyVisualProperty, updateCyVisualProperties`

Examples

```r
## Prepare used properties
## Visual property: Properties
vpPropertyP1 = createCyVisualPropertyProperties(c(NODE BORDER STROKE="SOLID"))
vpPropertyP2 = createCyVisualPropertyProperties(c(NODE BORDER WIDTH="1.5"))
vpPropertyP3 = createCyVisualPropertyProperties(c(NODE BORDER WIDTH="999"))

## Add two properties:
vpPropertyP4 = updateCyVisualProperty(vpPropertyP1, vpPropertyP2)
vpPropertyP4 = updateCyVisualProperty(vpPropertyP4, vpPropertyP3)

## Visual property: Dependencies
vpPropertyD1 = createCyVisualPropertyDependencies(c(nodeSizeLocked="false"))
vpPropertyD2 = createCyVisualPropertyDependencies(c(arrowColorMatchesEdge="true"))
vpPropertyD3 = createCyVisualPropertyDependencies(c(arrowColorMatchesEdge="false"))

## Add two dependencies:
vpPropertyD4 = updateCyVisualProperty(vpPropertyD1, vpPropertyD2)
vpPropertyD4 = updateCyVisualProperty(vpPropertyD4, vpPropertyD3)

## Visual property: Mappings
vpPropertyM1 = createCyVisualPropertyMappings(c(NODE_FILL_COLOR="CONTINUOUS"),
                                                  "COL=directed,T=boolean,K=0=true,V=0=ARROW")
vpPropertyM2 = createCyVisualPropertyMappings(c(EDGE_TARGET_ARROW SHAPE="DISCRETE"),
                                                  "TRIANGLE")
vpPropertyM3 = createCyVisualPropertyMappings(c(EDGE_TARGET_ARROW SHAPE="DISCRETE"),
                                                  "NONE")

## Add two mappings:
vpPropertyM4 = updateCyVisualProperty(vpPropertyM1, vpPropertyM2)
vpPropertyM4 = updateCyVisualProperty(vpPropertyM4, vpPropertyM3)

## Create visual property object
vpProperty1 = createCyVisualProperty(properties=list(vpPropertyP1,
                                                   vpPropertyP1),
                                       dependencies=list(vpPropertyD1,
                                                        vpPropertyD1, NA),
                                       dependencies=list(vpPropertyD3,
                                                        vpPropertyD1, NA),
                                       dependencies=list(vpPropertyM1,
                                                        vpPropertyM1, NA),
                                       dependencies=list(vpPropertyM4,
                                                        vpPropertyM4, NA))
```
updateEdgeAttributes

mappings=list(vpPropertyM1, NA, vpPropertyM1),
appliesTo = c(NA, NA, 1),
view = c(NA, 1, NA))

vpProperty2 = createCyVisualProperty(properties=vpPropertyP2,
dependencies=vpPropertyD2,
mappings=vpPropertyM2)

vpProperty3 = createCyVisualProperty(properties=vpPropertyP3,
dependencies=vpPropertyD3,
mappings=vpPropertyM3)

## add two visual property objects
vpProperty4 = updateCyVisualProperty(vpProperty1, vpProperty2)

## update values
updateCyVisualProperty(vpProperty4, vpProperty3)

## keep old values
updateCyVisualProperty(vpProperty4, vpProperty3,
replace = FALSE)

## keep old values
try(updateCyVisualProperty(vpProperty4, vpProperty3,
stopOnDuplicates = TRUE))

## =>ERROR:
## Elements of name (in properties<appliesTo=NA,view=NA>) must not contain duplicates!

updateEdgeAttributes  
Update edge attributes

Description

This functions add edge attributes in the form of a EdgeAttributes object to an RCX or an other 
EdgeAttributes object.

Usage

updateEdgeAttributes(
  x, 
  edgeAttributes, 
  replace = TRUE, 
  stopOnDuplicates = FALSE, 
  ... 
)
## S3 method for class 'EdgeAttributesAspect'
updateEdgeAttributes(
  x,
  edgeAttributes,
  replace = TRUE,
  stopOnDuplicates = FALSE,
  ...
)

## S3 method for class 'RCX'
updateEdgeAttributes(
  x,
  edgeAttributes,
  replace = TRUE,
  stopOnDuplicates = FALSE,
  checkReferences = TRUE,
  ...
)

### Arguments

- **x**
  - RCX or EdgeAttributes object; (to which the new edge attributes will be added)
- **edgeAttributes**
  - EdgeAttributes object; (the new aspect, that will be added)
- **replace**
  - logical; if existing values are updated (or ignored)
- **stopOnDuplicates**
  - logical; whether to stop, if duplicates in propertyOf and name (and subnet-workId if present) column are found
- **checkReferences**
  - logical; whether to check if references to other aspects are present in the RCX object

### Details

Edges may have additional attributes besides a name and a representation, and are represented as EdgeAttributes objects. EdgeAttributes objects can be added to an RCX or an other EdgeAttributes object. The propertyOf parameter references the Edges ids to which the attributes belong to. When adding an EdgeAttributes object to an RCX object, those ids must be present in the Edges aspect, otherwise an error is raised.

In the case, that a EdgeAttributes object is added to an other, or the RCX object already contains a EdgeAttributes object, some attributes might be present in both. By default, the attributes are updated with the values of the latest one. This can prevented by setting the replace parameter to FALSE, in that case only new attributes are added and the existing attributes remain untouched.

Furthermore, if duplicated attributes are considered as a preventable mistake, an error can be raised by setting stopOnDuplicates to TRUE. This forces the function to stop and raise an error, if duplicated attributes are present.
updateEdgeAttributes

Value

EdgeAttributes or RCX object with added node attributes

See Also

NodeAttributes, NetworkAttributes

Examples

```r
## For EdgeAttributesAspects:
## prepare some aspects:
edgeAttributes = createEdgeAttributes(
  propertyOf=c(0,0,0,0),
  name=c("A","A","B","B"),
  value=list(c("a1","a2"),
            "a with subnetwork",
            "b",
            "b with subnetwork"),
  isList=c(TRUE,FALSE,TRUE,FALSE),
  subnetworkId=c(NA,1,NA,1)
)

## A is updated, C is new
edgeAttributes2 = createEdgeAttributes(
  propertyOf=c(0,0,0),
  name=c("A","A","C"),
  value=list("new a",
            "new a with subnetwork",
            c(1,2)),
  subnetworkId=c(NA,1,NA)
)

## Simply update with new values
edgeAttributes3 = updateEdgeAttributes(edgeAttributes, edgeAttributes2)

## Ignore already present keys
edgeAttributes3 = updateEdgeAttributes(edgeAttributes, edgeAttributes2, replace=FALSE)

## Raise an error if duplicate keys are present
try(updateEdgeAttributes(edgeAttributes, edgeAttributes2, stopOnDuplicates=TRUE))

## =>ERROR:
## Elements of "propertyOf", "name" and "subnetworkId" (in updateEdgeAttributes)
## must not contain duplicates!

## For RCX
## prepare RCX object:
nodes = createNodes(name = c("a","b","c","d","e","f"))
edges = createEdges(source=c(1,2,0,0,0,2),
                   target=c(2,3,1,2,5,4))
rcx = createRCX(nodes, edges)
```
cySubNetworks = createCySubNetworks(
  id = c(1,2),
  nodes = list("all", c(1,2,3)),
  edges = list("all", c(0,2))
)
rcx = updateCySubNetworks(rcx, cySubNetworks)

## add the edge attributes
rcx = updateEdgeAttributes(rcx, edgeAttributes)

## add additional edge attributes and update existing
rcx = updateEdgeAttributes(rcx, edgeAttributes2)

## create edge attributes for a not existing edge...
edgeAttributes3 = createEdgeAttributes(propertyOf=9,
  name="A",
  value="a")

## ...and try to add them
try(updateEdgeAttributes(rcx, edgeAttributes3))
## =>ERROR:
## Provided IDs of "additionalAttributes$propertyOf" (in updateEdgeAttributes)
## don't exist in "rcx$edges$id"

---

updateEdges  Update edges

**Description**

This function adds edges in the form of an `Edges` object to another `Edges` or an `RCX` object.

**Usage**

updateEdges(x, edges, stopOnDuplicates = FALSE, keepOldIds = TRUE, ...)

## S3 method for class 'EdgesAspect'
updateEdges(x, edges, stopOnDuplicates = FALSE, keepOldIds = TRUE, ...)

## S3 method for class 'RCX'
updateEdges(
  x,
  edges,
  stopOnDuplicates = FALSE,
  keepOldIds = TRUE,
  checkReferences = TRUE,
  ...
)
Arguments

- **x**: RCX-object or Edges object; (to which the new Edges will be added)
- **edges**: Edges object; (the Edges, that will be added)
- **stopOnDuplicates**: logical (optional); whether to stop, if duplicates in id column are found, or re-assign ids instead.
- **keepOldIds**: logical (optional); if ids are re-assigned, the original ids are kept in the column oldId
- **...**: additional parameters
- **checkReferences**: logical; whether to check if references to other aspects are present in the RCX object

Details

When edges should be added to a Edges or a RCX-object object some conflicts may rise, since the aspects might use the same IDs. If the aspects do not share any IDs, the two aspects are simply combined. Otherwise, the IDs of the new edges are re-assinged continuing with the next available ID (i.e. maxId(edgesAspect) + 1 and maxId(rcx$edges) + 1, respectively).

To keep track of the changes, it is possible to keep the old IDs of the newly added edges in the automatically added column oldId. This can be omitted by setting keepOldIds to FALSE. Otherwise, if a re-assignment of the IDs is not desired, this can be prevented by setting stopOnDuplicates to TRUE. This forces the function to stop and raise an error, if duplicated IDs are present.

Value

Edges or RCX with added edges

Examples

```r
## create some edges
edges1 = createEdges(source=c(1,1,0), target=c(2,0,1))
edges2 = createEdges(id=c(3,2,4),
                     source=c(0,0,1),
                     target=c(1,2,2),
                     interaction=c("activates","inhibits", NA))

## simply add the edges and keep old ids
edges3 = updateEdges(edges1, edges2)

## add the edges
edges4 = updateEdges(edges1, edges2, keepOldIds=FALSE)

## force an error because of duplicated ids
try(updateEdges(edges1, edges2, stopOnDuplicates=TRUE))
## =>Error:
## Elements of "id" (in updateEdges) must not contain duplicates!
```
## Prepare an RCX object
rcx = createRCX(createNodes(name = c("EGFR","AKT1","WNT")))

## add edges to the RCX object
rcx = updateEdges(rcx, edges1)

## add new edges and don't keep old ids
rcx = updateEdges(rcx, edges2, keepOldIds=FALSE)

## force an error because of duplicated ids
try(updateEdges(rcx, edges2, stopOnDuplicates=TRUE))
## =>Error:
## Elements of "id" (in updateEdges) must not contain duplicates!

### updateMetaDataProperties

**Update meta-data properties**

**Description**

The **Meta-data** aspect contains meta-data about the aspects in the **RCX-object**. Properties that need to be fetched or updated independently of aspect data are added with this function.

**Usage**

updateMetaDataProperties(rcx, aspectName, property)

**Arguments**

- `rcx` **RCX object**;
- `aspectName` character; name of the aspect as displayed in **Meta-data** (e.g. "nodes")
- `property` named list; property as key-value pairs (empty list to remove all)

**Value**

**RCX object** with updated **Meta-data** aspect

**Examples**

## prepare RCX object:

```r
nodes = createNodes(name = c("a","b","c","d","e","f"))
edges = createEdges(source=c(1,2,0,0,0,2),
                     target=c(2,3,1,2,5,4))
rcx = createRCX(nodes, edges)
cySubNetworks = createCySubNetworks(
    id = c(1,2),
    nodes = list("all", c(1,2,3)),
    edges = list("all", c(0,2))
)
```
rcx = updateCySubNetworks(rcx, cySubNetworks)

## add properties for edges
updateMetaDataProperties(rcx,
    "edges",
    list(some="value",
         another="VALUE"))

## remove properties for edges
updateMetaDataProperties(rcx,
    "edges",
    list())

updateNetworkAttributes

Update network attributes

Description

This function adds network attributes in the form of a NetworkAttributes object to an RCX or another NetworkAttributes object.

Usage

updateNetworkAttributes(
  x,
  networkAttributes,
  replace = TRUE,
  stopOnDuplicates = FALSE,
  ...
)

## S3 method for class 'NetworkAttributesAspect'
updateNetworkAttributes(
  x,
  networkAttributes,
  replace = TRUE,
  stopOnDuplicates = FALSE,
  ...
)

## S3 method for class 'RCX'
updateNetworkAttributes(
  x,
  networkAttributes,
  replace = TRUE,
  stopOnDuplicates = FALSE,
  checkReferences = TRUE,
Arguments

- **x**: RCX object; (to which the new network attributes will be added)
- **networkAttributes**: NetworkAttributes object; (the new aspect, that will be added)
- **replace**: logical; if existing values are updated (or ignored)
- **stopOnDuplicates**: logical; whether to stop, if duplicates in *name* (and *subnetworkId* if present) column are found
- **...**: additional parameters
- **checkReferences**: logical; whether to check if references to other aspects are present in the RCX object

Details

Networks may have attributes, that are represented as NetworkAttributes objects. NetworkAttributes objects can be added to an RCX or an other NetworkAttributes object. In the case, that a NetworkAttributes object is added to an other, or the RCX object already contains a NetworkAttributes object, some attributes might be present in both. By default, the attributes are updated with the values of the latest one. This can prevented by setting the replace parameter to FALSE, in that case only new attributes are added and the existing attributes remain untouched.

Furthermore, if duplicated attributes are considered as a preventable mistake, an error can be raised by setting stopOnDuplicates to TRUE. This forces the function to stop and raise an error, if duplicated attributes are present.

Value

NetworkAttributes or RCX object with added network attributes

See Also

NetworkAttributes; NodeAttributes, EdgeAttributes

Examples

```r
## For NetworkAttributesAspects:
## prepare some aspects:
networkAttributes1 = createNetworkAttributes(
  name=c("A","A","B","B"),
  value=list(c("a1","a2"),
            "a with subnetwork",
            "b",
            "b with subnetwork"),
```
isList=c(TRUE,FALSE,TRUE,FALSE),
  subnetnetworkId=c(NA,1,NA,1)
)

## A is updated, C is new
networkAttributes2 = createNetworkAttributes(
  name=c("A","A","C"),
  value=list("new a",
             "new a with subnetwork",
             c(1,2)),
  subnetnetworkId=c(NA,1,NA)
)

## Simply update with new values
networkAttributes3 = updateNetworkAttributes(networkAttributes1, networkAttributes2)

## Ignore already present keys
networkAttributes3 = updateNetworkAttributes(networkAttributes1, networkAttributes2,
                                               replace=FALSE)

## Raise an error if duplicate keys are present
try(updateNetworkAttributes(networkAttributes1, networkAttributes2,
                              stopOnDuplicates=TRUE))
## =>ERROR:
## Provided IDs (name, subnetnetworkId) contain duplicates!

## For RCX
## prepare RCX object:
nodes = createNodes(name = c("a","b","c","d","e","f"))
edges = createEdges(source=c(1,2,0,0,0,2),
                    target=c(2,3,1,2,5,4))
rcx = createRCX(nodes, edges)
cySubNetworks = createCySubNetworks(
  id = c(1,2),
  nodes = list("all", c(1,2,3)),
  edges = list("all", c(0,2))
)
rcx = updateCySubNetworks(rcx, cySubNetworks)

## add the network attributes
rcx = updateNetworkAttributes(rcx, networkAttributes1)

## add additional network attributes and update existing
rcx = updateNetworkAttributes(rcx, networkAttributes2)

## create a relation with a not existing subnetwork...
networkAttributes3 = createNetworkAttributes(
  name="X",
  value="new x",
  subnetnetworkId=9)

## ...and try to add them
try(updateNetworkAttributes(rcx, networkAttributes3))
## =>ERROR:
## NetworkAttributesAspect$subnetworkId IDs don't exist in CySubNetworksAspect

updateNodeAttributes  Update node attributes

Description

This function adds node attributes in the form of a `NodeAttributes` object to an `RCX` or an other `NodeAttributes` object.

Usage

```
updateNodeAttributes(
  x,  
  nodeAttributes,  
  replace = TRUE,  
  stopOnDuplicates = FALSE,  
  ...
)
```

```
## S3 method for class 'NodeAttributesAspect'
updateNodeAttributes(
  x,  
  nodeAttributes,  
  replace = TRUE,  
  stopOnDuplicates = FALSE,  
  ...
)
```

```
## S3 method for class 'RCX'
updateNodeAttributes(
  x,  
  nodeAttributes,  
  replace = TRUE,  
  stopOnDuplicates = FALSE,  
  checkReferences = TRUE,  
  ...
)
```

Arguments

- `x`  
  RCX or `NodeAttributes` object; (to which the new node attributes will be added)

- `nodeAttributes`  
  `NodeAttributes` object; (the new aspect, that will be added)

- `replace`  
  logical; if existing values are updated (or ignored)
updateNodeAttributes

stopOnDuplicates
logical; whether to stop, if duplicates in propertyOf and name (and subnet-workId if present) columns are found

... additional parameters

cHECKReferences
logical; whether to check if references to other aspects are present in the RCX object

Details

Nodes may have additional attributes besides a name and a representation, and are represented as NodeAttributes objects. NodeAttributes objects can be added to an RCX object or an other NodeAttributes object. The propertyOf parameter references the node IDs to which the attributes belong to. When adding an NodeAttributes object to an RCX object, those IDs must be present in the Nodes aspect, otherwise an error is raised.

In the case, that a NodeAttributes object is added to an other, or the RCX object already contains a NodeAttributes object, some attributes might be present in both. By default, the attributes are updated with the values of the latest one. This can prevented setting the replace parameter to FALSE, in that case only new attributes are added and the existing attributes remain untouched.

Furthermore, if duplicated attributes are considered as a preventable mistake, an error can be raised by setting stopOnDuplicates to TRUE. This forces the function to stop and raise an error, if duplicated attributes are present.

Value

NodeAttributes or RCX object with added node attributes

See Also

EdgeAttributes, NetworkAttributes

Examples

## For NodeAttributesAspects:
## prepare some aspects:
nodeAttributes1 = createNodeAttributes(
  propertyOf=c(1,1,1,1),
  name=c("A","A","B","B"),
  value=list(c("a1","a2"),
            "a with subnetwork",
            "b",
            "b with subnetwork"),
  isList=c(TRUE,FALSE,TRUE,FALSE),
  subnetworkId=c(NA,1,NA,1)
)

## A is updated, C is new
nodeAttributes2 = createNodeAttributes(
  propertyOf=c(1,1,1,1),
  name=c("A","A","C"),
  value=list(c("a1","a2"),
            "a with subnetwork",
            "b with subnetwork"),
  isList=c(TRUE,FALSE,TRUE,FALSE),
  subnetworkId=c(NA,1,1,1)
)
value=list("new a",
    "new a with subnetwork",
    c(1,2)),
    subnetworkId=c(NA,1,NA)
)

## Simply update with new values
nodeAttributes3 = updateNodeAttributes(nodeAttributes1, nodeAttributes2)

## Ignore already present keys
nodeAttributes4 = updateNodeAttributes(nodeAttributes1, nodeAttributes2, replace=FALSE)

## Raise an error if duplicate keys are present
try(updateNodeAttributes(nodeAttributes1, nodeAttributes2, stopOnDuplicates=TRUE))

## =>ERROR:
## Elements of "propertyOf", "name" and "subnetworkId" (in addNodeAttributes)
## must not contain duplicates!

## For RCX
## prepare RCX object:
nodes = createNodes(name = c("a","b","c","d","e","f"))
edges = createEdges(source=c(1,2,0,0,0,2),
    target=c(2,3,1,2,5,4))
rcx = createRCX(nodes, edges)
cySubNetworks = createCySubNetworks(
    id = c(1,2),
    nodes = list("all", c(1,2,3)),
    edges = list("all", c(0,2)))
rcx = updateCySubNetworks(rcx, cySubNetworks)

## add the node attributes, even if no subnetworks are present
rcx = updateNodeAttributes(rcx, nodeAttributes1, checkReferences=FALSE)

## add the node attributes
rcx = updateNodeAttributes(rcx, nodeAttributes1)

## add additional node attributes and update existing
rcx = updateNodeAttributes(rcx, nodeAttributes2)

## create node attributes for a not existing node...
nodeAttributes3 = createNodeAttributes(propertyOf=9,
    name="A",
    value="a")

## ...and try to add them
try(updateNodeAttributes(rcx, nodeAttributes3))

## =>ERROR:
## Provided IDs of "additionalAttributes$propertyOf" (in addNodeAttributes)
## don't exist in "rcx$nodes$id"
updateNodes

Description

This functions add nodes in the form of a Nodes object to an other Nodes or an RCX-object.

Usage

updateNodes(x, nodes, stopOnDuplicates = FALSE, keepOldIds = TRUE)

## S3 method for class 'NodesAspect'
updateNodes(x, nodes, stopOnDuplicates = FALSE, keepOldIds = TRUE)

## S3 method for class 'RCX'
updateNodes(x, nodes, stopOnDuplicates = FALSE, keepOldIds = TRUE)

Arguments

- x: RCX-object or Nodes object; (to which the new Nodes will be added)
- nodes: Nodes object; (the Nodes, that will be added)
- stopOnDuplicates: logical (optional); whether to stop, if duplicates in id column are found, or re-assign ids instead.
- keepOldIds: logical (optional); if ids are re-assigned, the original ids are kept in the column oldId

Details

When nodes should be added to a Nodes or a RCX-object object some conflicts may rise, since the aspects might use the same IDs. If the aspects do not share any IDs, the two aspects are simply combined. Otherwise, the IDs of the new nodes are re-assinged continuing with the next available ID (i.e. maxId(nodesAspect) + 1 and maxId(rcx$nodes) + 1, respectively).

To keep track of the changes, it is possible to keep the old IDs of the newly added nodes in the automatically added column oldId. This can be omitted by setting keepOldIds to FALSE. Otherwise, if a re-assignment of the IDs is not desired, this can be prevented by setting stopOnDuplicates to TRUE. This forces the function to stop and raise an error, if duplicated IDs are present.

Value

Nodes or RCX object with added nodes
Examples

## create some nodes
nodes1 = createNodes(name = c("EGFR","AKT1","WNT"))
nodes2 = createNodes(name=c("CDK1", "CDK2", "CDK3"),
                    represents=c("HGNC:CDK1",
                                "Uniprot:P24941",
                                "Ensembl:ENSG00000250506"))

## simply add the nodes and keep old ids
nodes3 = updateNodes(nodes1, nodes2)

## add the nodes
nodes4 = updateNodes(nodes1, nodes2, keepOldIds=FALSE)

## force an error because of duplicated ids
try(updateNodes(nodes1, nodes2, stopOnDuplicates=TRUE))
## =>Error:
## Elements of "id" (in updateNodes) must not contain duplicates!

## create an RCX object with nodes
rcx = createRCX(nodes1)

## add additional nodes
rcx = updateNodes(rcx, nodes2, keepOldIds=FALSE)

## force an error becauses of duplicated ids
try(updateNodes(rcx, nodes2, stopOnDuplicates=TRUE))
## =>Error:
## Elements of "id" (in updateNodes) must not contain duplicates!

---

validate Validate RCX and its aspects

Description

Validate RCX objects and its aspects.

Usage

validate(x, verbose = TRUE)

## Default S3 method:
validate(x, verbose = TRUE)

## S3 method for class 'NodesAspect'
validate(x, verbose = TRUE)

## S3 method for class 'EdgesAspect'
validate(x, verbose = TRUE)
## Arguments

- **x**: object to validate; `RCX` object or an aspect
validate

verbose logical; whether to print the test results.

Details

Different tests are performed on aspects and the RCX network. This includes checks of the correct aspect structure, data types, uniqueness of IDs and attribute names, presence of NA values, and references between the aspects.

Value

logical; whether the object passed all tests.

Methods (by class)

• default: Default
• NodesAspect: Nodes
• EdgesAspect: Edges
• NodeAttributesAspect: Node attributes
• EdgeAttributesAspect: Edge attributes
• NetworkAttributesAspect: Network attributes
• CartesianLayoutAspect: Cartesian layout
• CyGroupsAspect: Cytoscape Groups
• CyVisualPropertiesAspect: Cytoscape Visual Properties
• CyVisualProperty: Cytoscape Visual Properties
• CyVisualPropertyProperties: Cytoscape visual property: Properties
• CyVisualPropertyDependencies: Cytoscape visual property: Dependencies
• CyVisualPropertyMappings: Cytoscape visual property: Mappings
• CyHiddenAttributesAspect: Cytoscape hidden attributes
• CyNetworkRelationsAspect: Cytoscape network relations
• CySubNetworksAspect: Cytoscape sub-networks
• CyTableColumnAspect: Cytoscape table column aspect
• RCX: The whole RCX object with all its aspects

Examples

## Read from a CX file
## reading the provided example network of the package
cxFile <- system.file("extdata","Imatinib-Inhibition-of-BCR-ABL-66a902f5-2022-11e9-bb6a-0ac135e8bacf.cx", package = "RCX")
rcx = readCX(cxFile)
## validate the network
validate(rcx)

## validate a single aspect
validate(rcx$nodes)

---

**visualize**

### Visualize a Network

**Description**

Visualize RCX and CX networks in RStudio or in an external browser.

**Usage**

```r
visualize(x, layout = NULL, openExternal = FALSE)
```

**Arguments**

- `x`: network; RCX or CX object
- `layout`: named character or list; e.g. `c(name="random")`
- `openExternal`: logical; whether to open in an external browser instead of the RStudio viewer

**Details**

This function uses the Java Script library used by the NDEx platform (https://ndexbio.org/) to visualize the RCX or CX network from `toCX`. In the first case, the RCX is converted to CX (JSON) using `toCX`.

By default the visualization is opened in RStudio in the **Viewer** panel. If this function is not executed in RStudio, the visualization is opened in the standard web-browser. This also can be forced from within RStudio using `openExternal`.


To define the layout of the network the coordinate from **CartesianLayout** are used to determine the location of the nodes. If this aspect is missing, or the the coordinates should be ignored, the `layout` parameter can be used to set a different layout.

`layout` follows therefore the definition of Cytoscape.js (see [https://js.cytoscape.org/#layouts](https://js.cytoscape.org/#layouts)). A simple definition can be setting only the `name` of the desired layout, e.g. `random`. Additional options can be passed as named list, where the values are passed without quoting. This allows for even passing Java Script functions to Cytoscape.js.

The visualization can also be saved as HTML file using the `writeHTML` function instead of this one.
writeCX

Write RCX to file

Value

NULL

See Also

rcxToJson, readCX, writeCX

Examples

```r
## prepare RCX
rcx = createRCX(
  createNodes(name = c("a", "b", "c")),
  createEdges(
    source=c(0,0,1),
    target=c(1,2,2)
  )
)

## visualize the network
visualize(rcx)

## force a different layout
visualize(rcx, c(name="cose"))

## force a different layout with Java Script parameters
visualize(rcx, layout = c(name="random",animate="true"))

## even pass a Java Script function
visualize(
  rcx,
  layout = c(
    name="random",
    animate="true",
    animateFilter="function ( node, i ){ return true; }"
  )
)

## open the visualization in an external browser
visualize(
  rcx,
  layout = c(name="cose"),
  openExternal = TRUE
)
```
Description

These function write an `RCX` object or a `CX` object to a file.

Usage

```r
writeCX(x, file, verbose = FALSE, pretty = FALSE)
```

Arguments

- `x`: `RCX` or `CX` object
- `file`: character; the name of the file to which the data are written
- `verbose`: logical; whether to print what is happening
- `pretty`: logical; adds indentation whitespace to JSON output. Can be TRUE/FALSE or a number specifying the number of spaces to indent. See `jsonlite::prettify()`

Value

- `file`: character; the name of the file to which the data were written

See Also

- `toCX`, `rcxToJson`, `readCX`

Examples

```r
NULL
```

---

### writeHTML

Save network visualization as HTML file

Description

Save an interactive single page visualization of `RCX` and `CX` networks as an HTML file containing all necessary Java Script.
Usage

```r
writeHTML(x, file, layout = NULL, verbose = FALSE)
```  
## S3 method for class 'RCX'

```r
writeHTML(x, file, layout = NULL, verbose = FALSE)
```  
## S3 method for class 'CX'

```r
writeHTML(x, file, layout = NULL, verbose = FALSE)
```  
Arguments

- `x`: network; RCX or CX object
- `file`: character; path, where the html file should be saved
- `layout`: named character or list; e.g. `c(name="random")`
- `verbose`: logical; whether to print what is happening

Details

This function uses the Java Script library used by the NDEx platform ([https://ndexbio.org/](https://ndexbio.org/)) to visualize the RCX or CX network. The RCX is therefore converted to CX (JSON) using `toCX`.


To define the layout of the network the coordinate from `CartesianLayout` are used to determine the location of the nodes. If this aspect is missing, or the the coordinates should be ignored, the `layout` parameter can be used to set a different layout.

`layout` follows therefore the definition of Cytoscape.js ([https://js.cytoscape.org/#layouts](https://js.cytoscape.org/#layouts)). A simple definition can be setting only the `name` of the desired layout, e.g. `random`. Additional options can be passed as named list, where the values are passed without quoting. This allows for even passing Java Script functions to Cytoscape.js.

To visualize the network in RStudio the `visualize` function can be used instead.

Value

- `file`: character; path, where the html file has been saved

See Also

- `rcxToJson`, `readCX`, `writeCX`

Examples

```r
## prepare RCX
rcx = createRCX(
  createNodes(name = c("a","b","c")),
  createEdges(
    source=c(0,0,1),
    target=c(1,2,2)
)
cx = toCX(rcx)

htmlFile = tempfile(fileext = ".html")

## save the html
writeHTML(rcx, htmlFile)

## or
writeHTML(cx, htmlFile)

## force a different layout
writeHTML(rcx, htmlFile, c(name="cose"))

## force a different layout with Java Script parameters
writeHTML(rcx, htmlFile, layout = c(name="random",animate="true"))

## even pass a Java Script function
writeHTML(
    rcx,
    htmlFile,
    layout = c(
        name="random",
        animate="true",
        animateFilter="function ( node, i ){ return true; }"
    )
)
)
Index

* Only
  .aspectClass, 5
* `aspectClasses`
  .aspectClass, 5
* are
  .aspectClass, 5
* aspects
  .aspectClass, 5
* class
  .aspectClass, 5
* datasets
  .aspectClass, 27
* defined
  .aspectClass, 5
* get
  .aspectClass, 5
* internal
  .addAspectNameToJson, 4
  .addAttributeData, 4
  .aspectClass, 5
  .convertRawList, 6
  .createAttributeAspect, 7
  .createCyVpPorD, 8
  .errorCodes, 9
  .filterBy, 11
  .format, 11
  .infer, 13
  .json2RDataType, 14
  .jsonL, 15
  .jsonV, 16
  .log, 16
  .mergeAttributesAspect, 17
  .mergeIdAspect, 18
  .modClass, 19
  .pasteC, 20
  .renameDF, 21
  .stop, 22
  .summaryAspect, 24
  .transformListLength<-, 25
  .transformVLD, 25
  .validateAttributesAspect, 26
  .validateCyVisualPropertyPandD, 27
  checks, 30
  convert-data-types-and-values, 34
  convert2json, 36
  dot_test, 58
  markAttributeColumn, 76
  markRefColumn, 77
* in
  .aspectClass, 5
* of
  .aspectClass, 5
* that
  .aspectClass, 5
* the
  .aspectClass, 5
  .addAspectNameToJson, 4
  .addAttributeData, 4
  .addClass (.modClass), 19
  .addClass<- (.modClass), 19
  .aspectClass, 5
  .checkAllClass (checks), 30
  .checkAllNumeric (checks), 30
  .checkAllNumericOrInDict (checks), 30
  .checkAnyNotNull (checks), 30
  .checkBContainsAllA (checks), 30
  .checkCharacter (checks), 30
  .checkClass (checks), 30
  .checkClassOneOf (checks), 30
  .checkIsId (checks), 30
  .checkIsUniqueId (checks), 30
  .checkList (checks), 30
  .checkLogical (checks), 30
  .checkNamed (checks), 30
  .checkNamedCharacter (checks), 30
  .checkNamedList (checks), 30
  .checkNamedLogical (checks), 30
  .checkNamedNumeric (checks), 30

146
.checkNoNa (checks), 30
.checkNonNeg (checks), 30
.checkNumeric (checks), 30
.checkRefPresent (checks), 30
.checkRefs (checks), 30
.checkSameLength (checks), 30
.checkUnique (checks), 30
.checkUniqueDF (checks), 30
.convert2json (convert2json), 36
.convertDataTypes
   (convert-data-types-and-values), 34
.convertRawList, 6
.convertValues
   (convert-data-types-and-values), 34
.createAttributeAspect, 7, 18, 19
.createCyVpPorD, 8
.elementsBContainsAllA (checks), 30
.elementsInDict (checks), 30
.elementsUnique (checks), 30
.elementsUniqueDF (checks), 30
.errorCodes, 9
.filterBy, 11
.format, 11
.formatComma (.format), 11
.formatData (.format), 11
.formatLog (.format), 11
.formatO (.format), 11
.formatParams (.format), 11
.formatQuote (.format), 11
.infer, 13
.inferDataType (.infer), 13
.inferIsList (.infer), 13
.json2RDataType, 14
.jsonL, 15
.jsonV, 16
.listAllNumeric (checks), 30
.listAllNumericOrInDict (checks), 30
.log, 16
.markAttributeValue<-
   (markAttributeValue), 76
.markRefColumn<- (markRefColumn), 77
.markReqRefColumn<- (markRefColumn), 77
.mergeAttributesAspect, 7, 17, 19
.mergeIdAspect, 7, 18, 19
.modClass, 19
.paramAnyNotNull (checks), 30
.paramClass (checks), 30
.paramIsOptionalId (checks), 30
.paramNamed (checks), 30
.paramNoNa (checks), 30
.paramNonNeg (checks), 30
.pasteC, 20
.removeClass (.modClass), 19
.removeClass<- (.modClass), 19
.renameDF, 21
.stop, 22
.summaryAspect, 24
.test_AllowedColumnsPresent (dot_test), 58
.test_AspectExist (dot_test), 58
.test_AtLeastOneColumnPresent
   (dot_test), 58
.test_AtLeastOneElementPresent
   (dot_test), 58
.test_ContainsNA (dot_test), 58
.test_DataTypeColumn (dot_test), 58
.test_ElementIsList (dot_test), 58
.test_ElementIsNumeric (dot_test), 58
.test_IdsInAspect (dot_test), 58
.test_IsCVPclass (dot_test), 58
.test_IsCharacter (dot_test), 58
.test_IsClass (dot_test), 58
.test_IsList (dot_test), 58
.test_IsLogical (dot_test), 58
.test_IsNamedList (dot_test), 58
.test_IsNumeric (dot_test), 58
.test_IsPos (dot_test), 58
.test_IsUnique (dot_test), 58
.test_IsUniqueInLists (dot_test), 58
.test_ListAllCharacter (dot_test), 58
.test_ListAllContainsNA (dot_test), 58
.test_ListAllNumeric (dot_test), 58
.test_ListAllNumericOrInDict
   (dot_test), 58
.test_ListAllOfClass (dot_test), 58
.test_ListAllOfCVPclass (dot_test), 58
.test_ListRequiredColumnsPresent
   (dot_test), 58
.test_NoMergeColumn (dot_test), 58
.test_OneNodePresent (dot_test), 58
.test_RequiredColumnsPresent (dot_test), 58
.test_ValuesInSet (dot_test), 58
.transformListLength<-, 25
transformVLD, 25
.validateAttributesAspect, 26
.validateCyVisualPropertyPandD, 27
.validateListOfCyVisualPropertyPandD (validateCyVisualPropertyPandD), 27

aspectClass2Name (Convert-Names-and-Classes), 35
aspectClasses, 27, 79, 93, 95, 98
aspectName2Class (Convert-Names-and-Classes), 35

base::print (), 39
cartesian layout, 69, 73
cartesianLayout, 29, 87, 101–103, 141, 144
cartesianLayout, checks, 30
convert-data-types-and-values, 34
Convert-Names-and-Classes, 35
countElements, 37, 78
createCartesianLayout (CartesianLayout), 29
createCyGroups (CyGroups), 40
createCyHiddenAttributes (CyHiddenAttributes), 41
createCyNetworkRelations (CyNetworkRelations), 43
createCySubNetworks (CySubNetworks), 45
createCyTableColumn (CyTableColumn), 46
createCyVisualProperties (CyVisualProperties), 47
createCyVisualProperty (CyVisualProperty), 50
createCyVisualPropertyDependencies (CyVisualPropertyDependencies), 52
createCyVisualPropertyMappings (CyVisualPropertyMappings), 54
createCyVisualPropertyProperties (CyVisualPropertyProperties), 56
createEdgeAttributes (EdgeAttributes), 62
createEdges (Edges), 64
createNetworkAttributes (NetworkAttributes), 80
createNodeAttributes (NodeAttributes), 82
createNodes (Nodes), 85
creartRCX (RCX-object), 87
custom-print, 38
CX, 143, 144
CyGroups, 40, 87, 104, 105
CyHiddenAttributes, 41, 87, 106–108
CyNetworkRelations, 29, 43, 45, 47, 87, 109–111, 114
CySubNetworks, 7, 29, 45, 88, 110, 112–114, 116, 119, 123
CyTableColumn, 46, 88, 115, 116
Cytoscape visual properties, 52, 54, 56, 118
CyVisualProperties, 37, 47, 48, 50, 53, 54, 57, 66, 87, 118, 119, 122
CyVisualPropertyDependencies, 8, 27, 48, 50, 51, 52, 53–57, 66, 119, 122–124
CyVisualPropertyMappings, 8, 48, 50–53, 54, 55–57, 66, 119, 122–124
dot_test, 58
dotTest, 68, 69, 72, 73
dot_id, 45
dot_ids, 40, 62
EdgeAttributes, 62, 81, 84, 87, 125–127, 132, 135
edgeAttributes, 69, 73
Edges, 40, 63, 64, 87, 105, 113, 126, 128, 129
edges, 69, 73
fromGraphNEL (graphNEL), 68
fromIgraph (Igraph), 72
getAspectClasses, 97
getAspectClasses (aspectClasses), 27
getCyVisualProperty, 49, 65, 120, 124
graph vertex attributes, 69
INDEX

graphNEL, 68, 68, 69, 73

hasIds, 70, 78
hasIds(), 38, 72, 78, 95, 97

idProperty, 71
idProperty(), 38, 71, 78, 95, 97

igraph, 69, 72

igraph graph attributes, 73
igraph vertex attributes, 73

igraph::as_graphnel(), 69

jsonlite, 75, 93

jsonlite::prettify(), 100, 143

jsonToRCX, 74, 92, 94

markAttributeColumn, 76

markRefColumn, 77

maxId, 77, 78, 105, 113, 129, 137

maxId(), 38, 71, 72, 78, 95, 97

Meta-data, 78

NetworkAttributes, 80, 84, 87, 127, 131, 132, 135

node, 29, 68, 72

node id, 45, 64

node ids, 29, 40, 83

NodeAttributes, 75, 81, 82, 87, 127, 132, 134, 135

nodeAttributes, 68, 69, 73

Nodes, 37, 40, 64, 69, 73, 83, 85, 87, 105, 113, 135, 137

nodes, 68, 69, 73, 87


parseJSON (readCX), 93

print.CartesianLayoutAspect (custom-print), 38

print.CyGroupsAspect (custom-print), 38

print.CyHiddenAttributesAspect (custom-print), 38

print.CyNetworkRelationsAspect (custom-print), 38

print.CySubNetworksAspect (custom-print), 38

print.CyTableColumnAspect (custom-print), 38

print.CyVisualPropertiesAspect (custom-print), 38

print.CyVisualProperty (custom-print), 38

print.CyVisualPropertyDependencies (custom-print), 38

print.CyVisualPropertyMappings (custom-print), 38

print.CyVisualPropertyProperties (custom-print), 38

print.EdgeAttributesAspect (custom-print), 38

print.EdgesAspect (custom-print), 38

print.MetaDataAspect (custom-print), 38

print.NetworkAttributesAspect (custom-print), 38

print.NodeAttributesAspect (custom-print), 38

print.NodesAspect (custom-print), 38

print.RCX (custom-print), 38

processCX, 75

processCX (readCX), 93


RCX-object, 87

rcxToJson, 76, 90, 101, 142–144

readCX, 76, 92, 93, 101, 142–144

readJSON (readCX), 93

referredBy, 95

referredBy(), 38, 71, 72, 78, 97

refersTo, 96

refersTo(), 38, 71, 72, 78, 95

setExtension, 27, 28, 97

subAspectClasses (aspectClasses), 27

subnetwork, 50, 69, 73

subnetwork id, 29, 44, 46, 62, 69, 73, 81, 83

subnetworks, 42, 44, 47, 62, 81, 83

summary, 40, 98

toCX, 76, 92, 100, 101, 141, 143, 144

toGraphNEL (graphNEL), 68

toIgraph (Igraph), 72

updateAspectClasses (aspectClasses), 27

updateCartesianLayout, 29, 101

updateCyGroups, 41, 104
updateCyHiddenAttributes, 42, 106
updateCyNetworkRelations, 44, 109
updateCySubNetworks, 45, 112
updateCyTableColumn, 47, 115
updateCyVisualProperty, 49, 51, 53, 56, 58, 67, 120, 122
updateEdgeAttributes, 63, 125
updateEdges, 65, 94, 128
updateMetaData (Meta-data), 78
updateMetaDataProperties, 79, 80, 130
updateNetworkAttributes, 81, 131
updateNodeAttributes, 84, 134
updateNodes, 86, 94, 137
validate, 138
vertex, 69, 73
visualize, 141, 144
writeCX, 76, 92, 94, 101, 142, 142, 144
writeHTML, 141, 143