Package 'AnnotationDbi'

December 29, 2018

Title Annotation Database Interface

Description Provides user interface and database connection code for annotation data packages using SQLite data storage.

Version 1.44.0

Encoding UTF-8

Author Hervé Pagès, Marc Carlson, Seth Falcon, Nianhua Li

Maintainer Bioconductor Package Maintainer <maintainer@bioconductor.org>

Depends R (>= 2.7.0), methods, utils, stats4, BiocGenerics (>= 0.23.1), Biobase (>= 1.17.0), IRanges

Imports DBI, RSQLite, S4Vectors (>= 0.9.25)

Suggests hgu95av2.db, GO.db, org.Sc.sgd.db, org.At.tair.db, KEGG.db, RUnit, TxDb.Hsapiens.UCSC.hg19.knownGene, hom.Hs.inp.db, org.Hs.eg.db, reactome.db, AnnotationForge, graph, EnsDb.Hsapiens.v75, BiocStyle, knitr

Collate 00RTobjs.R AllGenerics.R AllClasses.R unlist2.R utils.R SQL.R FlatBimap.R AnnDbObj-lowAPI.R Bimap.R GOTerms.R BimapFormatting.R Bimap-envirAPI.R flatten.R methods-AnnotationDb.R methods-SQLiteConnection.R methods-geneCentricDbs.R methods-geneCentricDbs-keys.R methods-ReactomeDb.R methods-Inparanoid.R methods-Inparanoid8.R loadDb.R createAnnObjs-utils.R createAnnObjs.NCBIORG_DBs.R createAnnObjs.NCBICHIP_DBs.R createAnnObjs.ORGANISM_DB.R createAnnObjs.YEASTCHIP_DB.R createAnnObjs.COELICOLOR_DB.R createAnnObjs.ARABIDOPSISCHIP_DB.R createAnnObjs.MALARIA_DB.R createAnnObjs.YEAST_DB.R createAnnObjs.YEASTNCBI_DB.R createAnnObjs.ARABIDOPSIS_DB.R createAnnObjs.GO_DB.R createAnnObjs.KEGG_DB.R createAnnObjs.INPARANOID_DB.R createAnnObjs.PFAM_DB.R AnnDbPkg-templates-common.R AnnDbPkg-checker.R print.probetable.R makeMap.R inpIDMapper.R test_AnnotationDbi_package.R

License Artistic-2.0

biocViews Annotation, Microarray, Sequencing, GenomeAnnotation

VignetteBuilder knitr

Video https://www.youtube.com/watch?v=8qvGNTVz3Ik

git_url https://git.bioconductor.org/packages/AnnotationDbi

git_branch RELEASE_3_8

git_last_commit ce191b0

git_last_commit_date 2018-10-30

Date/Publication 2018-12-29

R topics documented:

ACCNUM *Descriptions of available values for* `columns` *and* `keytypes`.

Description

This manual page enumerates the kinds of data represented by the values returned when the user calls `columns` or `keytypes`

Details

All the possible values for `columns` and `keytypes` are listed below. Users will have to actually use these methods to learn which of the following possible values actually apply in their case.

ACCNUM: GenBank accession numbers

ALIAS: Commonly used gene symbols

ARACYC: KEGG Identifiers for arabidopsis as indicated by aracyc

ARACYCENZYME: Aracyc enzyme names as indicated by aracyc

CHR: Chromosome (deprecated for Bioc > 3.1) For this information you should look at a TxDb or OrganismDb object and search for an appropriate field like TXCHROM, EXONCHROM or CDSCHROM. This information can also be retrieved from these objects using an appropriate range based accesor like transcripts, transcriptsBy etc.

CHRLOC: Chromosome and starting base of associated gene (deprecated for Bioc > 3.1) For this information you should look at a TxDb or OrganismDb object and search for an appropriate field like TXSTART etc. or even better use the associated range based accessors like transcripts or transcriptsBy to get back GRanges objects.

CHRLOCEND: Chromosome and ending base of associated gene (deprecated for Bioc > 3.1) For this information you should look at a TxDb or OrganismDb object and search for an appropriate field like TXEND etc. or even better use the associated range based accessors like transcripts or transcriptsBy to get back GRanges objects.

COMMON: Common name

DESCRIPTION: The description of the associated gene

ENSEMBL: The ensembl ID as indicated by ensembl

ENSEMBLPROT: The ensembl protein ID as indicated by ensembl

ENSEMBLTRANS: The ensembl transcript ID as indicated by ensembl

ENTREZID: Entrez gene Identifiers

ENZYME: Enzyme Commission numbers

EVIDENCE: Evidence codes for GO associations with a gene of interest

EVIDENCEALL: Evidence codes for GO (includes less specific terms)

GENENAME: The full gene name

GO: GO Identifiers associated with a gene of interest

GOALL: GO Identifiers (includes less specific terms)

INTERPRO: InterPro identifiers

IPI: IPI accession numbers

MAP: cytoband locations

OMIM: Online Mendelian Inheritance in Man identifiers

ONTOLOGY: For GO Identifiers, which Gene Ontology (BP, CC, or MF)

ONTOLOGYALL: Which Gene Ontology (BP, CC, or MF), (includes less specific terms)

ORF: Yeast ORF Identifiers

PATH: KEGG Pathway Identifiers

PFAM: PFAM Identifiers

PMID: Pubmed Identifiers

PROBEID: Probe or manufacturer Identifiers for a chip package

PROSITE: Prosite Identifiers

REFSEQ: Refseq Identifiers

SGD: Saccharomyces Genome Database Identifiers

SMART: Smart Identifiers

SYMBOL: The official gene symbol

TAIR: TAIR Identifiers

UNIGENE: Unigene Identifiers

UNIPROT: Uniprot Identifiers

To get the latest information about the date stamps and source URLS for the data used to make an annotation package, please use the metadata method as shown in the example below.

Unless otherwise indicated above, the majority of the data for any one package is taken from the source indicated by either it's name (if it's an org package) OR from the name of it's associated org package. So for example, org.Hs.eg.db is using "eg" in the name to indicate that most of the data in that package comes from NCBI entrez gene based data. And org.At.tair.db uses data that primarily comes from tair. For chip packages, the relevant information is the organism package that they depend on. So for example, hgu95av2.db depends on org.Hs.eg.db, and is thus primarily based on NCBI entrez gene ID information.

Author(s)

Marc Carlson

Examples

```
library(hgu95av2.db)
## List the possible values for columns
columns(hgu95av2.db)
## List the possible values for keytypes
keytypes(hgu95av2.db)
## get some values back
keys <- head(keys(hgu95av2.db))
keys
select(hgu95av2.db, keys=keys, columns=c("SYMBOL","PFAM"),
keytype="PROBEID")

## More infomation about the dates and original sources for these data:
metadata(hgu95av2.db)
```

AnnDbObj-objects *AnnDbObj objects*

Description

The AnnDbObj class is the most general container for storing any kind of SQLite-based annotation data.

Details

Many classes in AnnotationDbi inherit directly or indirectly from the AnnDbObj class. One important particular case is the AnnDbBimap class which is the lowest class in the AnnDbObj hierarchy to also inherit the Bimap interface.

Accessor-like methods

In the code snippets below, `x` is an AnnDbObj object.

`dbconn(x)`: Return a connection object to the SQLite DB containing `x`'s data.

`dbfile(x)`: Return the path (character string) to the SQLite DB (file) containing `x`'s data.

dbmeta(x, name): Print the value of metadata whose name is 'name'. Also works if x is a DBIConnection object.

dbschema(x, file="", show.indices=FALSE): Print the schema definition of the SQLite DB. Also works if x is a DBIConnection object.

The file argument must be a connection, or a character string naming the file to print to (see the file argument of the cat function for the details).

The CREATE INDEX statements are not shown by default. Use show.indices=TRUE to get them.

dbInfo(x): Prints other information about the SQLite DB. Also works if x is a DBIConnection object.

See Also

dbConnect, dbListTables, dbListFields, dbGetQuery, Bimap

Examples

```
library("hgu95av2.db")

dbconn(hgu95av2ENTREZID)              # same as hgu95av2_dbconn()
dbfile(hgu95av2ENTREZID)              # same as hgu95av2_dbfile()

dbmeta(hgu95av2_dbconn(), "ORGANISM")
dbmeta(hgu95av2_dbconn(), "DBSCHEMA")
dbmeta(hgu95av2_dbconn(), "DBSCHEMAVERSION")

library("DBI")
dbListTables(hgu95av2_dbconn())       #lists all tables on connection

## If you use dbSendQuery instead of dbGetQuery
## (NOTE: for ease of use, this is defintitely NOT reccomended)
## Then you may need to know how to list results objects
dbListResults(hgu95av2_dbconn())      #for listing results objects


## You can also list the fields by using this connection
dbListFields(hgu95av2_dbconn(), "probes")
dbListFields(hgu95av2_dbconn(), "genes")
dbschema(hgu95av2ENTREZID)        # same as hgu95av2_dbschema()
## According to the schema, the probes._id column references the genes._id
## column. Note that in all tables, the "_id" column is an internal id with
## no biological meaning (provided for allowing efficient joins between
## tables).
## The information about the probe to gene mapping is in probes:
dbGetQuery(hgu95av2_dbconn(), "SELECT * FROM probes LIMIT 10")
## This mapping is in fact the ENTREZID map:
toTable(hgu95av2ENTREZID)[1:10, ] # only relevant columns are retrieved

dbInfo(hgu95av2GO)                # same as hgu95av2_dbInfo()

##Advanced example:
##Sometimes you may wish to join data from across multiple databases at
##once:
## In the following example we will attach the GO database to the
```

```
## hgu95av2 database, and then grab information from separate tables
## in each database that meet a common criteria.
library(hgu95av2.db)
library("GO.db")
attachSql <- paste('ATTACH "', GO_dbfile(), '" as go;', sep = "")
dbGetQuery(hgu95av2_dbconn(), attachSql)
sql <- 'SELECT  DISTINCT a.go_id AS "hgu95av2.go_id",
         a._id AS "hgu95av2._id",
         g.go_id AS "GO.go_id", g._id AS "GO._id",
         g.term, g.ontology, g.definition
         FROM go_bp_all AS a, go.go_term AS g
         WHERE a.go_id = g.go_id LIMIT 10;'
data <- dbGetQuery(hgu95av2_dbconn(), sql)
data
## For illustration purposes, the internal id "_id" and the "go_id"
## from both tables is included in the output.  This makes it clear
## that the "go_ids" can be used to join these tables but the internal
## ids can NOT.  The internal IDs (which are always shown as _id) are
## suitable for joins within a single database, but cannot be used
## across databases.
```

AnnDbPkg-checker *Check the SQL data contained in an SQLite-based annotation package*

Description

Check the SQL data contained in an SQLite-based annotation package.

Usage

```
checkMAPCOUNTS(pkgname)
```

Arguments

pkgname	The name of the SQLite-based annotation package to check.

Author(s)

H. Pages

See Also

AnnDbPkg-maker

Examples

```
checkMAPCOUNTS("org.Sc.sgd.db")
```

AnnotationDb-objects *AnnotationDb objects and their progeny, methods etc.*

Description

AnnotationDb is the virtual base class for all annotation packages. It contain a database connection and is meant to be the parent for a set of classes in the Bioconductor annotation packages. These classes will provide a means of dispatch for a widely available set of select methods and thus allow the easy extraction of data from the annotation packages.

select, columns and keys are used together to extract data from an AnnotationDb object (or any object derived from the parent class). Examples of classes derived from the AnnotationDb object include (but are not limited to): ChipDb, OrgDb GODb, InparanoidDb and ReactomeDb.

columns shows which kinds of data can be returned for the AnnotationDb object.

keytypes allows the user to discover which keytypes can be passed in to select or keys and the keytype argument.

keys returns keys for the database contained in the AnnotationDb object . This method is already documented in the keys manual page but is mentioned again here because it's usage with select is so intimate. By default it will return the primary keys for the database, but if used with the keytype argument, it will return the keys from that keytype.

select will retrieve the data as a data.frame based on parameters for selected keys columns and keytype arguments. Users should be warned that if you call select and request columns that have multiple matches for your keys, select will return a data.frame with one row for each possible match. This has the effect that if you request multiple columns and some of them have a many to one relationship to the keys, things will continue to multiply accordingly. So it's not a good idea to request a large number of columns unless you know that what you are asking for should have a one to one relationship with the initial set of keys. In general, if you need to retrieve a column (like GO) that has a many to one relationship to the original keys, it is most useful to extract that separately.

mapIds gets the mapped ids (column) for a set of keys that are of a particular keytype. Usually returned as a named character vector.

saveDb will take an AnnotationDb object and save the database to the file specified by the path passed in to the file argument.

loadDb takes a .sqlite database file as an argument and uses data in the metadata table of that file to return an AnnotationDb style object of the appropriate type.

species shows the genus and species label currently attached to the AnnotationDb objects database.

dbfile gets the database file associated with an object.

dbconn gets the datebase connection associated with an object.

taxonomyId gets the taxonomy ID associated with an object (if available).

Usage

```
columns(x)
keytypes(x)
keys(x, keytype, ...)
select(x, keys, columns, keytype, ...)
mapIds(x, keys, column, keytype, ..., multiVals)
saveDb(x, file)
loadDb(file, packageName=NA)
```

Arguments

x	the AnnotationDb object. But in practice this will mean an object derived from an AnnotationDb object such as a OrgDb or ChipDb object.
keys	the keys to select records for from the database. All possible keys are returned by using the keys method.
columns	the columns or kinds of things that can be retrieved from the database. As with keys, all possible columns are returned by using the columns method.
keytype	the keytype that matches the keys used. For the select methods, this is used to indicate the kind of ID being used with the keys argument. For the keys method this is used to indicate which kind of keys are desired from keys
column	the column to search on (for mapIds). Different from columns in that it can only have a single element for the value
...	other arguments. These include: **pattern:** the pattern to match (used by keys) **column:** the column to search on. This is used by keys and is for when the thing you want to pattern match is different from the keytype, or when you want to simply want to get keys that have a value for the thing specified by the column argument. **fuzzy:** TRUE or FALSE value. Use fuzzy matching? (this is used with pattern by the keys method)
multiVals	What should mapIds do when there are multiple values that could be returned? Options include: **first:** This value means that when there are multiple matches only the 1st thing that comes back will be returned. This is the default behavior **list:** This will just returns a list object to the end user **filter:** This will remove all elements that contain multiple matches and will therefore return a shorter vector than what came in whenever some of the keys match more than one value **asNA:** This will return an NA value whenever there are multiple matches **CharacterList:** This just returns a SimpleCharacterList object **FUN:** You can also supply a function to the multiVals argument for custom behaviors. The function must take a single argument and return a single value. This function will be applied to all the elements and will serve a 'rule' that for which thing to keep when there is more than one element. So for example this example function will always grab the last element in each result: `last <- function(x){x[[length(x)]]}`
file	an sqlite file path. A string the represents the full name you want for your sqlite database and also where to put it.
packageName	for internal use only

Value

keys,columns and keytypes each return a character vector or possible values. select returns a data.frame.

Author(s)

Marc Carlson

See Also

keys, dbConnect, dbListTables, dbListFields, dbGetQuery, Bimap

Examples

```
require(hgu95av2.db)
## display the columns
columns(hgu95av2.db)
## get the 1st 6 possible keys
keys <- head( keys(hgu95av2.db) )
keys
## lookup gene symbol and unigene ID for the 1st 6 keys
select(hgu95av2.db, keys=keys, columns = c("SYMBOL","UNIGENE"))

## get keys based on unigene
keyunis <- head( keys(hgu95av2.db, keytype="UNIGENE") )
keyunis
## list supported key types
keytypes(hgu95av2.db)
## lookup gene symbol and unigene ID based on unigene IDs by setting
## the keytype to "UNIGENE" and passing in unigene keys:
select(hgu95av2.db, keys=keyunis, columns = c("SYMBOL","UNIGENE"),
       keytype="UNIGENE")

keys <- head(keys(hgu95av2.db, 'ENTREZID'))
## get a default result (captures only the 1st element)
mapIds(hgu95av2.db, keys=keys, column='ALIAS', keytype='ENTREZID')
## or use a different option
mapIds(hgu95av2.db, keys=keys, column='ALIAS', keytype='ENTREZID',
    multiVals="CharacterList")
## Or define your own function
last <- function(x){x[[length(x)]]}
mapIds(hgu95av2.db, keys=keys, column='ALIAS', keytype='ENTREZID',
    multiVals=last)

## For other ways to access the DB, you can use dbfile() or dbconn() to extract
dbconn(hgu95av2.db)
dbfile(hgu95av2.db)

## Try to retrieve an associated taxonomyId
taxonomyId(hgu95av2.db)
```

Bimap *Bimap objects and the Bimap interface*

Description

What we usually call "annotation maps" are in fact Bimap objects. In the following sections we present the bimap concept and the Bimap interface as it is defined in AnnotationDbi.

Display methods

In the code snippets below, x is a Bimap object.

`show(x)`: Display minimal information about Bimap object `x`.

`summary(x)`: Display a little bit more information about Bimap object `x`.

The bimap concept

A bimap is made of:

```
- 2 sets of objects: the left objects and the right objects.
  All the objects have a name and this name is unique in
  each set (i.e. in the left set and in the right set).
  The names of the left (resp. right) objects are called the
  left (resp. right) keys or the Lkeys (resp. the Rkeys).


- Any number of links (edges) between the left and right
  objects. Note that the links can be tagged. In our model,
  for a given bimap, either none or all the links are tagged.
```

In other words, a bimap is a bipartite graph.

Here are some examples:

```
1. bimap B1:

   4 left objects (Lkeys): "a", "b", "c", "d"
   3 objects on the right (Rkeys): "A", "B", "C"

   Links (edges):
    "a" <--> "A"
    "a" <--> "B"
    "b" <--> "A"
    "d" <--> "C"

   Note that:
     - There can be any number of links starting from or ending
       at a given object.
     - The links in this example are untagged.


2. bimap B2:

   4 left objects (Lkeys): "a", "b", "c", "d"
   3 objects on the right (Rkeys): "A", "B", "C"

   Tagged links (edges):
     "a" <-"x"-> "A"
     "a" <-"y"-> "B"
     "b" <-"x"-> "A"
     "d" <-"x"-> "C"
     "d" <-"y"-> "C"

   Note that there are 2 links between objects "d" and "C":
   1 with tag "x" and 1 with tag "y".
```

Flat representation of a bimap

The flat representation of a bimap is a data frame. For example, for B1, it is:

```
left  right
   a      A
   a      B
   b      A
   d      C
```

If in addition the right objects have 1 multivalued attribute, for example, a numeric vector:

```
A <-- c(1.2, 0.9)
B <-- character(0)
C <-- -1:1
```

then the flat representation of B1 becomes:

```
left  right  Rattrib1
   a      A       1.2
   a      A       0.9
   a      B        NA
   b      A       1.2
   b      A       0.9
   d      C        -1
   d      C         0
   d      C         1
```

Note that now the number of rows is greater than the number of links!

AnnDbBimap and FlatBimap objects

An AnnDbBimap object is a bimap whose data are stored in a data base. A FlatBimap object is a bimap whose data (left keys, right keys and links) are stored in memory (in a data frame for the links). Conceptually, AnnDbBimap and FlatBimap objects are the same (only their internal representation differ) so it's natural to try to define a set of methods that make sense for both (so they can be manipulated in a similar way). This common interface is the Bimap interface.

Note that both AnnDbBimap and FlatBimap objects have a read-only semantic: the user can subset them but cannot change their data.

The "flatten" generic

```
flatten(x) converts AnnDbBimap object x into FlatBimap
object y with no loss of information
```

Note that a FlatBimap object can't be converted into an AnnDbBimap object (well, in theory maybe it could be, but for now the data bases we use to store the data of the AnnDbBimap objects are treated as read-only). This conversion from AnnDbBimap to FlatBimap is performed by the "flatten" generic function (with methods for AnnDbBimap objects only).

Property0

The "flatten" generic plays a very useful role when we need to understand or explain exactly what a given Bimap method f will do when applied to an AnnDbBimap object. It's generally easier to explain what it does on a FlatBimap object and then to just say "and it does the same thing on an AnnDbBimap object". This is exactly what Property0 says:

```
for any AnnDbBimap object x, f(x) is expected to be
indentical to f(flatten(x))
```

Of course, this implies that the f method for AnnDbBimap objects return the same type of object than the f method for FlatBimap objects. In this sense, the "revmap" and "subset" Bimap methods are particular because they are expected to return an object of the same class as their argument x, so f(x) can't be identical to f(flatten(x)). For these methods, Property0 says:

```
for any AnnDbBimap object x, flatten(f(x)) is expected to
be identical to f(flatten(x))
```

Note to the AnnotationDbi maintainers/developpers: the `checkProperty0` function (AnnDbPkg-checker.R file) checks that Property0 is satisfied on all the AnnDbBimap objects defined in a given package (FIXME: checkProperty0 is currently broken).

The Bimap interface in AnnotationDbi

The full documentation for the methods of the Bimap interface is splitted into 4 man pages: Bimap-direction, Bimap-keys and Bimap-toTable.

See Also

Bimap-direction, Bimap-keys, Bimap-toTable, BimapFormatting, Bimap-envirAPI

Examples

```
library(hgu95av2.db)
ls(2)
hgu95av2GO # calls the "show" method
summary(hgu95av2GO)
hgu95av2GO2PROBE # calls the "show" method
summary(hgu95av2GO2PROBE)
```

Bimap-direction — *Methods for getting/setting the direction of a Bimap object, and undirected methods for getting/counting/setting its keys*

Description

These methods are part of the Bimap interface (see `?Bimap` for a quick overview of the Bimap objects and their interface).

They are divided in 2 groups: (1) methods for getting or setting the direction of a Bimap object and (2) methods for getting, counting or setting the left or right keys (or mapped keys only) of a Bimap object. Note that all the methods in group (2) are undirected methods i.e. what they return does NOT depend on the direction of the map (more on this below).

Usage

```
## Getting or setting the direction of a Bimap object
direction(x)
direction(x) <- value
revmap(x, ...)

## Getting, counting or setting the left or right keys (or mapped
## keys only) of a Bimap object
Lkeys(x)
Rkeys(x)
Llength(x)
Rlength(x)
mappedLkeys(x)
mappedRkeys(x)
count.mappedLkeys(x)
count.mappedRkeys(x)
Lkeys(x) <- value
Rkeys(x) <- value
## S4 method for signature 'Bimap'
subset(x, Lkeys = NULL, Rkeys = NULL, drop.invalid.keys = FALSE)
## S4 method for signature 'AnnDbBimap'
subset(x, Lkeys = NULL, Rkeys = NULL, drop.invalid.keys = FALSE,
    objName = NULL)
```

Arguments

x	A Bimap object.
value	A single integer or character string indicating the new direction in `direction(x) <- value`. A character vector containing the new keys (must be a subset of the current keys) in `Lkeys(x) <- value` and `Rkeys(x) <- value`.
Lkeys, Rkeys, drop.invalid.keys, objName, ...	Extra arguments for `revmap` and `subset`. Extra argument for `revmap` can be: `objName` The name to give to the reversed map (only supported if `x` is an AnnDbBimap object). Extra arguments for `subset` can be: `Lkeys` The new Lkeys. `Rkeys` The new Rkeys. `drop.invalid.keys` If `drop.invalid.keys=FALSE` (the default), an error will be raised if the new Lkeys or Rkeys contain invalid keys i.e. keys that don't belong to the current Lkeys or Rkeys. If `drop.invalid.keys=TRUE`, invalid keys are silently dropped. `objName` The name to give to the submap (only supported if `x` is an AnnDbBimap object).

Details

All Bimap objects have a direction which can be left-to-right (i.e. the mapping goes from the left keys to the right keys) or right-to-left (i.e. the mapping goes from the right keys to the left keys).

A Bimap object x that maps from left to right is considered to be a direct map. Otherwise it is considered to be an indirect map (when it maps from right to left).

direction returns 1 on a direct map and -1 otherwise.

The direction of x can be changed with direction(x) <- value where value must be 1 or -1. An easy way to reverse a map (i.e. to change its direction) is to do direction(x) <- - direction(x), or, even better, to use revmap(x) which is actually the recommended way for doing it.

The Lkeys and Rkeys methods return respectively the left and right keys of a Bimap object. Unlike the keys method (see ?keys for more information), these methods are direction-independent i.e. what they return does NOT depend on the direction of the map. Such methods are also said to be "undirected methods" and methods like the keys method are said to be "directed methods".

All the methods described below are also "undirected methods".

Llength(x) and Rlength(x) are equivalent to (but more efficient than) length(Lkeys(x)) and length(Rkeys(x)), respectively.

The mappedLkeys (or mappedRkeys) method returns the left keys (or right keys) that are mapped to at least one right key (or one left key).

count.mappedLkeys(x) and count.mappedRkeys(x) are equivalent to (but more efficient than) length(mappedLkeys(x)) and length(mappedRkeys(x)), respectively. These functions give overall summaries, if you want to know how many Rkeys correspond to a given Lkey you can use the nhit function.

Lkeys(x) <- value and Rkeys(x) <- value are the undirected versions of keys(x) <- value (see ?keys for more information) and subset(x, Lkeys=new_Lkeys, Rkeys=new_Rkeys) is provided as a convenient way to reduce the sets of left and right keys in one single function call.

Value

1L or -1L for direction.

A Bimap object of the same subtype as x for revmap and subset.

A character vector for Lkeys, Rkeys, mappedLkeys and mappedRkeys.

A single non-negative integer for Llength, Rlength, count.mappedLkeys and count.mappedRkeys.

Author(s)

H. Pages

See Also

Bimap, Bimap-keys, BimapFormatting, Bimap-envirAPI, nhit

Examples

```
library(hgu95av2.db)
ls(2)
x <- hgu95av2GO
x
summary(x)
direction(x)

length(x)
Llength(x)
Rlength(x)
```

```
keys(x)[1:4]
Lkeys(x)[1:4]
Rkeys(x)[1:4]

count.mappedkeys(x)
count.mappedLkeys(x)
count.mappedRkeys(x)

mappedkeys(x)[1:4]
mappedLkeys(x)[1:4]
mappedRkeys(x)[1:4]

y <- revmap(x)
y
summary(y)
direction(y)

length(y)
Llength(y)
Rlength(y)

keys(y)[1:4]
Lkeys(y)[1:4]
Rkeys(y)[1:4]

## etc...

## Get rid of all unmapped keys (left and right)
z <- subset(y, Lkeys=mappedLkeys(y), Rkeys=mappedRkeys(y))
```

Bimap-envirAPI *Environment-like API for Bimap objects*

Description

These methods allow the user to manipulate any Bimap object as if it was an environment. This environment-like API is provided for backward compatibility with the traditional environment-based maps.

Usage

```
ls(name, pos = -1L, envir = as.environment(pos), all.names = FALSE,
   pattern, sorted = TRUE)
exists(x, where, envir, frame, mode, inherits)
get(x, pos, envir, mode, inherits)
#x[[i]]
#x$name

## Converting to a list
mget(x, envir, mode, ifnotfound, inherits)
eapply(env, FUN, ..., all.names, USE.NAMES)
#contents(object, all.names)
```

```
## Additional convenience method
sample(x, size, replace=FALSE, prob=NULL, ...)
```

Arguments

name	A Bimap object for ls. A key as a literal character string or a name (possibly backtick quoted) for x$name.
pos, all.names, USE.NAMES, where, frame, mode, inherits	Ignored.
envir	Ignored for ls. A Bimap object for mget, get and exists.
pattern	An optional regular expression. Only keys matching 'pattern' are returned.
x	The key(s) to search for for exists, get and mget. A Bimap object for [[and x$name. A Bimap object or an environment for sample.
i	Single key specifying the map element to extract.
ifnotfound	A value to be used if the key is not found. Only NA is currently supported.
env	A Bimap object.
FUN	The function to be applied (see original eapply for environments for the details).
...	Optional arguments to FUN.
size	Non-negative integer giving the number of map elements to choose.
replace	Should sampling be with replacement?
prob	A vector of probability weights for obtaining the elements of the map being sampled.
sorted	logical(1). When TRUE (default), return primary keys in sorted order.

See Also

ls, exists, get, mget, eapply, contents, sample, BimapFormatting, Bimap

Examples

```
library(hgu95av2.db)
x <- hgu95av2CHRLOC

ls(x)[1:3]
exists(ls(x)[1], x)
exists("titi", x)
get(ls(x)[1], x)
x[[ls(x)[1]]]
x$titi # NULL

mget(ls(x)[1:3], x)
eapply(x, length)
contents(x)

sample(x, 3)
```

Bimap-keys *Methods for manipulating the keys of a Bimap object*

Description

These methods are part of the Bimap interface (see ?Bimap for a quick overview of the Bimap objects and their interface).

Usage

```
#length(x)
isNA(x)
mappedkeys(x)
count.mappedkeys(x)
keys(x) <- value
#x[i]
```

Arguments

x	A Bimap object. If the method being caled is keys(x), then x can also be a AnnotationDb object or one of that objects progeny.
value	A character vector containing the new keys (must be a subset of the current keys).
i	A character vector containing the keys of the map elements to extract.

Details

keys(x) returns the set of all valid keys for map x. For example, keys(hgu95av2GO) is the set of all probe set IDs for chip hgu95av2 from Affymetrix.

Please Note that in addition to Bimap objest, keys(x) will also work for AnnotationDb objects and related objects such as OrgDb and ChipDb objects.

Note also that the double bracket operator [[for Bimap objects is guaranteed to work only with a valid key and will raise an error if the key is invalid. (See ?`Bimap-envirAPI` for more information about this operator.)

length(x) is equivalent to (but more efficient than) length(keys(x)).

A valid key is not necessarily mapped ([[will return an NA on an unmapped key).

isNA(x) returns a logical vector of the same length as x where the TRUE value is used to mark keys that are NOT mapped and the FALSE value to mark keys that ARE mapped.

mappedkeys(x) returns the subset of keys(x) where only mapped keys were kept.

count.mappedkeys(x) is equivalent to (but more efficient than) length(mappedkeys(x)).

Two (almost) equivalent forms of subsetting a Bimap object are provided: (1) by setting the keys explicitely and (2) by using the single bracket operator [for Bimap objects. Let's say the user wants to restrict the mapping to the subset of valid keys stored in character vector mykeys. This can be done either with keys(x) <- mykeys (form (1)) or with y <- x[mykeys] (form (2)). Please note that form (1) alters object x in an irreversible way (the original keys are lost) so form (2) should be preferred.

All the methods described on this pages are "directed methods" i.e. what they return DOES depend on the direction of the Bimap object that they are applied to (see ?direction for more information about this).

Value

A character vector for `keys` and `mappedkeys`.

A single non-negative integer for `length` and `count.mappedkeys`.

A logical vector for `isNA`.

A Bimap object of the same subtype as `x` for `x[i]`.

Author(s)

H. Pages

See Also

Bimap, Bimap-envirAPI, Bimap-toTable, BimapFormatting, AnnotationDb, select, columns

Examples

```
library(hgu95av2.db)
x <- hgu95av2GO
x
length(x)
count.mappedkeys(x)
x[1:3]
links(x[1:3])

## Keep only the mapped keys
keys(x) <- mappedkeys(x)
length(x)
count.mappedkeys(x)
x # now it is a submap

## The above subsetting can also be achieved with
x <- hgu95av2GO[mappedkeys(hgu95av2GO)]

## mappedkeys() and count.mappedkeys() also work with an environment
## or a list
z <- list(k1=NA, k2=letters[1:4], k3="x")
mappedkeys(z)
count.mappedkeys(z)

## retrieve the set of primary keys for the ChipDb object named 'hgu95av2.db'
keys <- keys(hgu95av2.db)
head(keys)
```

Bimap-toTable *Methods for manipulating a Bimap object in a data-frame style*

Description

These methods are part of the Bimap interface (see ?Bimap for a quick overview of the Bimap objects and their interface).

Usage

```
  ## Extract all the columns of the map (links + right attributes)
  toTable(x)
  nrow(x)
  ncol(x)
  #dim(x)
  ## S4 method for signature 'FlatBimap'
head(x, ...)
  ## S4 method for signature 'FlatBimap'
tail(x, ...)

  ## Extract only the links of the map
  links(x)
  count.links(x)
  nhit(x)

  ## Col names and col metanames
  colnames(x, do.NULL=TRUE, prefix="col")
  colmetanames(x)
  Lkeyname(x)
  Rkeyname(x)
  keyname(x)
  tagname(x)
  Rattribnames(x)
  Rattribnames(x) <- value
```

Arguments

x	A Bimap object (or a list or an environment for nhit).
...	Further arguments to be passed to or from other methods (see head or tail for the details).
do.NULL	Ignored.
prefix	Ignored.
value	A character vector containing the names of the new right attributes (must be a subset of the current right attribute names) or NULL.

Details

toTable(x) turns Bimap object x into a data frame (see section "Flat representation of a bimap" in ?Bimap for a short introduction to this concept). For simple maps (i.e. no tags and no right attributes), the resulting data frame has only 2 columns, one for the left keys and one for the right keys, and each row in the data frame represents a link (or edge) between a left and a right key. For maps with tagged links (i.e. a tag is associated to each link), toTable(x) has one additional colmun for the tags and there is still one row per link. For maps with right attributes (i.e. a set of attributes is associated to each right key), toTable(x) has one additional colmun per attribute. So for example if x has tagged links and 2 right attributes, toTable(x) will have 5 columns: one for the left keys, one for the right keys, one for the tags, and one for each right attribute (always the rightmost columns). Note that if at least one of the right attributes is multivalued then more than 1 row can be needed to represent the same link so the number of rows in toTable(x) can be strictly greater than the number of links in the map.

nrow(x) is equivalent to (but more efficient than) nrow(toTable(x)).

`ncol(x)` is equivalent to (but more efficient than) `ncol(toTable(x))`.

`colnames(x)` is equivalent to (but more efficient than) `colnames(toTable(x))`. Columns are named accordingly to the names of the SQL columns where the data are coming from. An important consequence of this that they are not necessarily unique.

`colmetanames(x)` returns the metanames for the column of `x` that are not right attributes. Valid column metanames are `"Lkeyname"`, `"Rkeyname"` and `"tagname"`.

`Lkeyname`, `Rkeyname`, `tagname` and `Rattribnames` return the name of the column (or columns) containing the left keys, the right keys, the tags and the right attributes, respectively.

Like `toTable(x)`, `links(x)` turns `x` into a data frame but the right attributes (if any) are dropped. Note that dropping the right attributes produces a data frame that has eventually less columns than `toTable(x)` and also eventually less rows because now exactly 1 row is needed to represent 1 link.

`count.links(x)` is equivalent to (but more efficient than) `nrow(links(x))`.

`nhit(x)` returns a named integer vector indicating the number of "hits" for each key in `x` i.e. the number of links that start from each key.

Value

A data frame for `toTable` and `links`.

A single integer for `nrow`, `ncol` and `count.links`.

A character vector for `colnames`, `colmetanames` and `Rattribnames`.

A character string for `Lkeyname`, `Rkeyname` and `tagname`.

A named integer vector for `nhit`.

Author(s)

H. Pages

See Also

Bimap, BimapFormatting, Bimap-envirAPI

Examples

```
library(GO.db)
x <- GOSYNONYM
x
toTable(x)[1:4, ]
toTable(x["GO:0007322"])
links(x)[1:4, ]
links(x["GO:0007322"])

nrow(x)
ncol(x)
dim(x)
colnames(x)
colmetanames(x)
Lkeyname(x)
Rkeyname(x)
tagname(x)
Rattribnames(x)

links(x)[1:4, ]
```

```
count.links(x)

y <- GOBPCHILDREN
nhy <- nhit(y) # 'nhy' is a named integer vector
identical(names(nhy), keys(y)) # TRUE
table(nhy)
sum(nhy == 0) # number of GO IDs with no children
names(nhy)[nhy == max(nhy)] # the GO ID(s) with the most direct children

## Some sanity check
sum(nhy) == count.links(y) # TRUE

## Changing the right attributes of the GOSYNONYM map (advanced
## users only)
class(x) # GOTermsAnnDbBimap
as.list(x)[1:3]
colnames(x)
colmetanames(x)
tagname(x) # untagged map
Rattribnames(x)
Rattribnames(x) <- Rattribnames(x)[3:1]
colnames(x)
class(x) # AnnDbBimap
as.list(x)[1:3]
```

BimapFormatting *Formatting a Bimap as a list or character vector*

Description

These functions format a Bimap as a list or character vector.

Usage

```
## Formatting as a list
as.list(x, ...)

## Formatting as a character vector
#as.character(x, ...)
```

Arguments

x	A Bimap object.
...	Further arguments are ignored.

Author(s)

H. Pages

See Also

Bimap, Bimap-envirAPI

Examples

```
library(hgu95av2.db)
as.list(hgu95av2CHRLOC)[1:9]
as.list(hgu95av2ENTREZID)[1:9]
as.character(hgu95av2ENTREZID)[1:9]
```

`createSimpleBimap` *Creates a simple Bimap from a SQLite database in an situation that is external to AnnotationDbi*

Description

This function allows users to easily make a simple Bimap object for extra tables etc that they may wish to add to their annotation packages. For most Bimaps, their definition is stored inside of AnnotationDbi. The addition of this function is to help ensure that this does not become a limitation, by allowing simple extra Bimaps to easily be defined external to AnnotationDbi. Usually, this will be done in the zzz.R source file of a package so that these extra mappings can be seemlessly integrated with the rest of the package. For now, this function assumes that users will want to use data from just one table.

Usage

```
createSimpleBimap(tablename, Lcolname, Rcolname, datacache, objName,
objTarget)
```

Arguments

`tablename`	The name of the database table to grab the mapping information from.
`Lcolname`	The field name from the database table. These will become the Lkeys in the final mapping.
`Rcolname`	The field name from the database table. These will become the Rkeys in the final mapping.
`datacache`	The datacache object should already exist for every standard Annotation package. It is not exported though, so you will have to access it with ::: . It is needed to provide the connection information to the function.
`objName`	This is the name of the mapping.
`objTarget`	This is the name of the thing the mapping goes with. For most uses, this will mean the package name that the mapping belongs with.

Examples

```
##You simply have to call this function to create a new mapping.  For
##example, you could have created a mapping between the gene_name and
##the symbols fields from the gene_info table contained in the hgu95av2
##package by doing this:
library(hgu95av2.db)
hgu95av2NAMESYMBOL <- createSimpleBimap("gene_info",
                                        "gene_name",
                                        "symbol",
                                        hgu95av2.db:::datacache,
                                        "NAMESYMBOL",
```

```
                                    "hgu95av2.db")
```

GOFrame *GOFrame and GOAllFrame objects*

Description

These objects each contain a data frame which is required to be composed of 3 columns. The 1st column are GO IDs. The second are evidence codes and the 3rd are the gene IDs that match to the GO IDs using those evidence codes. There is also a slot for the organism that these anotations pertain to.

Details

The GOAllFrame object can only be generated from a GOFrame object and its contructor method does this automatically from a GOFrame argument. The purpose of these objects is to create a safe way for annotation data about GO from non-traditional sources to be used for analysis packages like GSEABase and eventually GOstats.

Examples

```
## Make up an example
genes = c(1,10,100)
evi = c("ND","IEA","IDA")
GOIds = c("GO:0008150","GO:0008152","GO:0001666")
frameData = data.frame(cbind(GOIds,evi,genes))

library(AnnotationDbi)
frame=GOFrame(frameData,organism="Homo sapiens")
allFrame=GOAllFrame(frame)

getGOFrameData(allFrame)
```

GOID *Descriptions of available values for* columns *and* keytypes *for GO.db.*

Description

This manual page enumerates the kinds of data represented by the values returned when the user calls columns or keytypes

Details

All the possible values for columns and keytypes are listed below.

GOID: GO Identifiers

DEFINITION: The definition of a GO Term

ONTOLOGY: Which of the three Gene Ontologies (BP, CC, or MF)

TERM: The actual GO term

To get the latest information about the date stamps and source URLS for the data used to make an annotation package, please use the metadata method as shown in the example below.

Author(s)

Marc Carlson

Examples

```
library(GO.db)
## List the possible values for columns
columns(GO.db)
## List the possible values for keytypes
keytypes(GO.db)
## get some values back
keys <- head(keys(GO.db))
keys
select(GO.db, keys=keys, columns=c("TERM","ONTOLOGY"),
keytype="GOID")

## More infomation about the dates and original sources for these data:
metadata(GO.db)
```

GOTerms-class *Class "GOTerms"*

Description

A class to represent Gene Ontology nodes

Objects from the Class

Objects can be created by calls of the form `GOTerms(GOId, term, ontology, definition, synonym, secondary)`. GOId, term, and ontology are required.

Slots

`GOID`: Object of class `"character"` A character string for the GO id of a primary node.

`Term`: Object of class `"character"` A character string that defines the role of gene product corresponding to the primary GO id.

`Ontology`: Object of class `"character"` Gene Ontology category. Can be MF - molecular function, CC - cellular component, or BP - biological process.

`Definition`: Object of class `"character"` Further definition of the ontology of the primary GO id.

`Synonym`: Object of class `"character"` other ontology terms that are considered to be synonymous to the primary term attached to the GO id (e.g. "type I programmed cell death" is a synonym of "apoptosis"). Synonymous here can mean that the synonym is an exact synonym of the primary term, is related to the primary term, is broader than the primary term, is more precise than the primary term, or name is related to the term, but is not exact, broader or narrower.

`Secondary`: Object of class `"character"` GO ids that are secondary to the primary GO id as results of merging GO terms so that One GO id becomes the primary GO id and the rest become the secondary.

Methods

GOID `signature(object = "GOTerms")`: The get method for slot GOID.

Term `signature(object = "GOTerms")`: The get method for slot Term.

Ontology `signature(object = "GOTerms")`: The get method for slot Ontology.

Definition `signature(object = "GOTerms")`: The get method for slot Definition.

Synonym `signature(object = "GOTerms")`: The get method for slot Synonym.

Secondary `signature(object = "GOTerms")`: The get method for slot Secondary.

show `signature(x = "GOTerms")`: The method for pretty print.

Note

GOTerms objects are used to represent primary GO nodes in the SQLite-based annotation data package GO.db

References

http://www.geneontology.org/

See Also

makeGOGraph shows how to make GO mappings into graphNEL objects.

Examples

```
  gonode <- new("GOTerms", GOID="GO:1234567", Term="Test", Ontology="MF",
                           Definition="just for testing")
  GOID(gonode)
  Term(gonode)
  Ontology(gonode)

  ##Or you can just use these methods on a GOTermsAnnDbBimap
## Not run: ##I want to show an ex., but don't want to require GO.db
  require(GO.db)
  FirstTenGOBimap <- GOTERM[1:10] ##grab the 1st ten
  Term(FirstTenGOBimap)

  ##Or you can just use GO IDs directly
  ids = keys(FirstTenGOBimap)
  Term(ids)

## End(Not run)
```

HOMO_SAPIENS *Descriptions of available values for* columns *and* keytypes *for inparanoid packages.*

Description

When the user calls columns or keytypes for an inparanoid package, the columns and keytypes methods will give the full genus and species names of all the organisms that are available.

Details

All the possible values for `columns` and `keytypes` are listed below.

ACYRTHOSIPHON_PISUM: the pea aphid

AEDES_AEGYPTI: a mosquito that can spread the dengue fever, Chikungunya and yellow fever viruses, and other diseases

ANOPHELES_GAMBIAE: a mosquito notorious as a vector for malaria

APIS_MELLIFERA: the western honey bee

ARABIDOPSIS_THALIANA: the thale cress

ASPERGILLUS_FUMIGATUS: a fungus that causes disease in immunodeficient individuals

BATRACHOCHYTRIUM_DENDROBATIDIS: a chytrid fungus that causes the disease chytridiomycosis

BOMBYX_MORI: the silk worm

BOS_TAURUS: domestic cattle

BRANCHIOSTOMA_FLORIDAE: a lancelet (amphioxus)

BRUGIA_MALAYI: a nematode (roundworm), one of the three causative agents of lymphatic filariasis

CAENORHABDITIS_BRENNERI: a small nematode, closely related to the model organism Caenorhabditis elegans

CAENORHABDITIS_BRIGGSAE: a small nematode, closely related to Caenorhabditis elegans

CAENORHABDITIS_ELEGANS: a small nematode

CAENORHABDITIS_JAPONICA: a gonochoristic (male-female) species related to C. elegans

CAENORHABDITIS_REMANEI: a species of nematode (gonochoristic)

CANDIDA_ALBICANS: a diploid fungus that grows both as yeast and filamentous cells and a causal agent of opportunistic oral and genital infections in humans

CANDIDA_GLABRATA: a haploid yeast of the genus Candida

CANIS_FAMILIARIS: domestic dog

CAPITELLA_SPI: a polychaete worm

CAVIA_PORCELLUS: Guinea pig

CHLAMYDOMONAS_REINHARDTII: a single celled green alga

CIONA_INTESTINALIS: a urochordata (sea squirt), a tunicate widely distributed in Northern European waters

CIONA_SAVIGNYI: a urochordata (sea squirt)

COCCIDIOIDES_IMMITIS: a pathogenic fungus that resides in the soil

COPRINOPSIS_CINEREUS: a species of mushroom

CRYPTOCOCCUS_NEOFORMANS: an encapsulated yeast that can live in both plants and animals

CRYPTOSPORIDIUM_HOMINIS: an obligate parasite of humans that can colonize the gastrointestinal tract

CRYPTOSPORIDIUM_PARVUM: one of several protozoal species that cause cryptosporidiosis, a parasitic disease of the mammalian intestinal tract

CULEX_PIPIENS: the common house mosquito

CYANIDIOSCHYZON_MEROLAE: a an algae that is the main organism in red tide

DANIO_RERIO: the zebrafish

DAPHNIA_PULEX: the most common species of water flea

DEBARYOMYCES_HANSENII: a yeast that tolerates high concentrations of salt and is related to yeasts that cause disease, including Candida albicans

DICTYOSTELIUM_DISCOIDEUM: a species of soil-living amoeba, AKA a slime mold

DROSOPHILA_ANANASSAE: a fruit fly

DROSOPHILA_GRIMSHAWI: a fruit fly

DROSOPHILA_MELANOGASTER: a fruit fly

DROSOPHILA_MOJAVENSIS: a fruit fly

DROSOPHILA_PSEUDOOBSCURA: a fruit fly

DROSOPHILA_VIRILIS: a fruit fly

DROSOPHILA_WILLISTONI: a fruit fly

ENTAMOEBA_HISTOLYTICA: an anaerobic parasitic protozoan

EQUUS_CABALLUS: domestic horse

ESCHERICHIA_COLIK12: a laboratory strain of coliform bacteria

FUSARIUM_GRAMINEARUM: a fungus that attacks cereal grains

GALLUS_GALLUS: domsticated chicken

GASTEROSTEUS_ACULEATUS: three spined stickleback fish

GIARDIA_LAMBLIA: a flagellated protozoan parasite

HELOBDELLA_ROBUSTA: a leech

IXODES_SCAPULARIS: the black legged deer tick, a vector for lyme disease

KLUYVEROMYCES_LACTIS: yeast commonly used for genetic studies

LEISHMANIA_MAJOR: a species of Leishmania, associated with zoonotic cutaneous leishmaniasis

LOTTIA_GIGANTEA: a species of sea snail, a true limpet, a marine gastropod mollusc

MACACA_MULATTA: the rhesus Macaque

MAGNAPORTHE_GRISEA: rice blast fungus

MONODELPHIS_DOMESTICA: grey short tailed opossum

MONOSIGA_BREVICOLLIS: a marine choanoflagellate

MUS_MUSCULUS: lab mouse

NASONIA_VITRIPENNIS: a small pteromalid parasitoid wasp

NEMATOSTELLA_VECTENSIS: the starlet sea anemone

NEUROSPORA_CRASSA: a type of red bread mould

ORNITHORHYNCHUS_ANATINUS: the platypus

ORYZA_SATIVA: rice

ORYZIAS_LATIPES: medaka fish

OSTREOCOCCUS_TAURI: a unicellular coccoid or spherically shaped green alga

PAN_TROGLODYTES: chimp

PEDICULUS_HUMANUS: a species of lice that infects humans

PHYSCOMITRELLA_PATENS: a moss (Bryophyta) used as a model organism for studies on plant evolution

PHYTOPHTHORA_RAMORUM: the oomycete plant pathogen (sudden oak death)

PHYTOPHTHORA_SOJAE: an oomycete and a soil-borne plant pathogen that causes stem and root rot of soybean

PLASMODIUM_FALCIPARUM: a protozoan parasite that causes malaria

PLASMODIUM_VIVAX: a protozoal parasite and a human pathogen that causes a more benign malaria

PONGO_PYGMAEUS: the Bornean orangutan

POPULUS_TRICHOCARPA: black cottonwood; also known as western balsam poplar or California poplar

PRISTIONCHUS_PACIFICUS: a diplogastrid nematode

PUCCINIA_GRAMINIS: stem, black or cereal rusts

RATTUS_NORVEGICUS: common lab rat

RHIZOPUS_ORYZAE: a fungus that lives worldwide in dead organic matter. An opportunistic human pathogen

SACCHAROMYCES_CEREVISIAE: brewers yeast

SCHISTOSOMA_MANSONI: a significant parasite of humans, a trematode that is one of the major agents of the disease schistosomiasis

SCHIZOSACCHAROMYCES_POMBE: fission yeast

SCLEROTINIA_SCLEROTIORUM: an omnivorous fungal plant pathogen

SORGHUM_BICOLOR: sorghum

STAGONOSPORA_NODORUM: a fungal leaf spot disease

STRONGYLOCENTROTUS_PURPURATUS: the purple sea urchin

TAKIFUGU_RUBRIPES: Japanese pufferfish

TETRAHYMENA_THERMOPHILA: a single celled cilliate

TETRAODON_NIGROVIRIDIS: green spotted pufferfish (fresh water)

THALASSIOSIRA_PSEUDONANA: a species of marine centric diatom

THEILERIA_ANNULATA: a tickborne protozoan pathogen which is a major cause of livestock disease in sub-tropical regions

THEILERIA_PARVA: a parasitic protozoan, that causes East Coast fever (theileriosis) in cattle

TRIBOLIUM_CASTANEUM: the red flour beetle

TRICHOMONAS_VAGINALIS: an anaerobic, flagellated protozoan

TRICHOPLAX_ADHAERENS: Trichoplax adhaerens represents the simplest known animal, with the smallest known animal genome

TRYPANOSOMA_CRUZI: a species of parasitic euglenoid trypanosomes. This species causes the trypanosomiasis diseases in humans and animals in America.

USTILAGO_MAYDIS: a pathogenic plant fungus that causes smut disease on maize

XENOPUS_TROPICALIS: Western clawed frog

YARROWIA_LIPOLYTICA: Yarrowia lipolytica is a "non-conventional" species of yeast, often used in genetic research because it differs from other well-studied species

To get the latest information about the date stamps and source URLS for the data used to make an annotation package, please use the metadata method as shown in the example below.

Author(s)

Marc Carlson

Examples

```
library(hom.Hs.inp.db)
## List the possible values for columns
columns(hom.Hs.inp.db)
## List the possible values for keytypes
keytypes(hom.Hs.inp.db)
## get some values back
keys <- head(keys(hom.Hs.inp.db, keytype="HOMO_SAPIENS"))
keys
select(hom.Hs.inp.db, keys=keys, columns=c("BOS_TAURUS","EQUUS_CABALLUS"),
keytype="HOMO_SAPIENS")

## More infomation about the dates and original sources for these data:
metadata(hom.Hs.inp.db)
```

`inpIDMapper` *Convenience functions for mapping IDs through an appropriate set of annotation packages*

Description

These are a set of convenience functions that attempt to take a list of IDs along with some addional information about what those IDs are, what type of ID you would like them to be, as well as some information about what species they are from and what species you would like them to be from and then attempts to the simplest possible conversion using the organism and possible inparanoid annotation packages. By default, this function will drop ambiguous matches from the results. Please see the details section for more information about the parameters that can affect this. If a more complex treatment of how to handle multiple matches is required, then it is likely that a less convenient approach will be necessary.

Usage

```
inpIDMapper(ids, srcSpecies, destSpecies, srcIDType="UNIPROT",
destIDType="EG", keepMultGeneMatches=FALSE, keepMultProtMatches=FALSE,
keepMultDestIDMatches = TRUE)

intraIDMapper(ids, species, srcIDType="UNIPROT", destIDType="EG",
keepMultGeneMatches=FALSE)

idConverter(ids, srcSpecies, destSpecies, srcIDType="UNIPROT",
destIDType="EG", keepMultGeneMatches=FALSE, keepMultProtMatches=FALSE,
keepMultDestIDMatches = TRUE)
```

Arguments

`ids`	a list or vector of original IDs to match

`srcSpecies` The original source species in in paranoid format. In other words, the 3 letters of the genus followed by 2 letters of the species in all caps. Ie. 'HOMSA' is for Homo sapiens etc.

`destSpecies` the destination species in inparanoid format

`species` the species involved

`srcIDType` The source ID type written exactly as it would be used in a mapping name for an eg package. So for example, 'UNIPROT' is how the uniprot mappings are always written, so we keep that convention here.

`destIDType` the destination ID, written the same way as you would write the srcIDType. By default this is set to "EG" for entrez gene IDs

`keepMultGeneMatches`
Do you want to try and keep the 1st ID in those ambiguous cases where more than one protein is suggested? (You probably want to filter them out - hence the default is FALSE)

`keepMultProtMatches`
Do you want to try and keep the 1st ID in those ambiguous cases where more than one protein is suggested? (default = FALSE)

`keepMultDestIDMatches`
If you have mapped to a destination ID OTHER than an entrez gene ID, then it is possible that there may be multiple answers. Do you want to keep all of these or only return the 1st one? (default = TRUE)

Details

inpIDMapper - This is a convenience function for getting an ID from one species mapped to an ID type of your choice from another organism of your choice. The only mappings used to do this are the mappings that are scored as 100 according to the inparanoid algorithm. This function automatically tries to join IDs by using FIVE different mappings in the sequence that follows:

1) initial IDs -> src organism Entrez Gene IDs 2) src organism Entrez Gene IDs -> sre organism Inparanoid ID 3) src organism Inparanoid ID -> dest organism Inparanoid ID 4) dest organism Inparanoid ID -> dest organism Entrez Gene ID 5) dest organism Entrez Gene ID -> final destination organism ID

You can simplify this mapping as a series of steps like this:

srcIDs —> srcEGs —> srcInp —> destInp —> destEGs —> destIDs (1) (2) (3) (4) (5)

There are two steps in this process where multiple mappings can really interfere with getting a clear answer. It's no coincidence that these are also adjacent to the two places where we have to tie the identity to a single gene for each organism. When this happens, any ambiguity is confounding. Preceding step \#2, it is critical that we only have ONE entrez gene ID per initial ID, and the parameter keepMultGeneMatches can be used to toggle whether to drop any ambiguous matches (the default) or to keep the 1st one in the hope of getting an additional hit. A similar thing is done preceding step \#4, where we have to be sure that the protein IDs we are getting back have all mapped to only one gene. We allow you to use the keepMultProtMatches parameter to make the same kind of decision as in step \#2, again, the default is to drop anything that is ambiguous.

intraIDMapper - This is a convenience function to map within an organism and so it has a much simpler job to do. It will either map through one mapping or two depending whether the source ID or destination ID is a central ID for the relevant organism package. If the answer is neither, then two mappings will be needed.

idConverter - This is mostly for convenient usage of these functions by developers. It is just a wrapper function that can pass along all the parameters to the appropriate function (intraIDMapper

or inpIDMapper). It decides which function to call based on the source and destination organism. The disadvantage to using this function all the time is just that more of the parameters have to be filled out each time.

Value

a list where the names of each element are the elements of the original list you passed in, and the values are the matching results. Elements that do not have a match are not returned. If you want things to align you can do some bookeeping.

Author(s)

Marc Carlson

Examples

```
## Not run:
  ## This has to be in a dontrun block because otherwise I would have to
  ## expand the DEPENDS field for AnnotationDbi
  ## library("org.Hs.eg.db")
  ## library("org.Mm.eg.db")
  ## library("org.Sc.eg.db")
  ## library("hom.Hs.inp.db")
  ## library("hom.Mm.inp.db")
  ## library("hom.Sc.inp.db")

  ##Some IDs just for the example
  library("org.Hs.eg.db")
  ids = as.list(org.Hs.egUNIPROT)[10000:10500] ##get some ragged IDs
  ## Get entrez gene IDs (default) for uniprot IDs mapping from human to mouse.
  MouseEGs = inpIDMapper(ids, "HOMSA", "MUSMU")
  ##Get yeast uniprot IDs in exchange for uniprot IDs from human
  YeastUPs = inpIDMapper(ids, "HOMSA", "SACCE", destIDType="UNIPROT")
  ##Get yeast uniprot IDs but only return one ID per initial ID
 YeastUPSingles = inpIDMapper(ids, "HOMSA", "SACCE", destIDType="UNIPROT", keepMultDestIDMatches = FALSE)

  ##Test out the intraIDMapper function:
  HumanEGs = intraIDMapper(ids, species="HOMSA", srcIDType="UNIPROT",
  destIDType="EG")
  HumanPATHs = intraIDMapper(ids, species="HOMSA", srcIDType="UNIPROT",
  destIDType="PATH")

  ##Test out the wrapper function
  MousePATHs = idConverter(MouseEGs, srcSpecies="MUSMU", destSpecies="MUSMU",
  srcIDType="EG", destIDType="PATH")
  ##Convert from Yeast uniprot IDs to Human entrez gene IDs.
  HumanEGs = idConverter(YeastUPSingles, "SACCE", "HOMSA")


## End(Not run)
```

KEGGFrame *KEGGFrame objects*

Description

These objects each contain a data frame which is required to be composed of 2 columns. The 1st column are KEGG IDs. The second are the gene IDs that match to the KEGG IDs. There is also a slot for the organism that these anotations pertain to. `getKEGGFrameData` is just an accessor method and returns the data.frame contained in the KEGGFrame object and is mostly used by other code internally.

Details

The purpose of these objects is to create a safe way for annotation data about KEGG from non-traditional sources to be used for analysis packages like GSEABase and eventually Category.

Examples

```
## Make up an example
genes = c(2,9,9,10)
KEGGIds = c("04610","00232","00983","00232")
frameData = data.frame(cbind(KEGGIds,genes))

library(AnnotationDbi)
frame=KEGGFrame(frameData,organism="Homo sapiens")

getKEGGFrameData(frame)
```

---

makeGOGraph *A convenience function to generate graphs based on the GO.db package*

Description

`makeGOGraph` is a function to quickly convert any of the three Gene Ontologies in GO.db into a graphNEL object where each edge is given a weight of 1.

Usage

```
makeGOGraph(ont = c("bp","mf","cc"))
```

Arguments

`ont`	Specifies the ontology: "cc", "bp" or "mf".

Author(s)

Marc Carlson

See Also

GOTerms

Examples

```
## makes a GO graph from the CC ontology
f <- makeGOGraph("cc")
```

---

make_eg_to_go_map *Create GO to Entrez Gene maps for chip-based packages*

Description

Create a new map object mapping Entrez ID to GO (or vice versa) given a chip annotation data package.

This is a temporary solution until a more general pluggable map solution comes online.

Usage

```
make_eg_to_go_map(chip)
```

Arguments

chip	The name of the annotation data package.

Value

Either a Go3AnnDbMap or a RevGo3AnnDbMap.

Author(s)

Seth Falcon and Herve Pages

Examples

```
library("hgu95av2.db")

eg2go = make_eg_to_go_map("hgu95av2.db")
sample(eg2go, 2)

go2eg = make_go_to_eg_map("hgu95av2.db")
sample(go2eg, 2)
```

orgPackageName *Org package contained in annotation object*

Description

Get the name of the org package used by an annotation resource object.

NOTE: This man page is for the `orgPackageName` *S4 generic function* defined in the **AnnotationDbi** package. Bioconductor packages can define specific methods for annotation objects not supported by the default method.

Usage

```
orgPackageName(x, ...)
```

Arguments

`x`	An annotation resource object.
`...`	Additional arguments.

Value

A `character(1)` vector indicating the org package name.

Specific methods defined in Bioconductor packages should behave as consistently as possible with the default method.

print.probetable *Print method for probetable objects*

Description

Prints class(x), nrow(x) and ncol(x), but not the elements of x. The motivation for having this method is that methods from the package `base` such as `print.data.frame` will try to print the values of all elements of x, which can take inconveniently much time and screen space if x is large.

Usage

```
## S3 method for class 'probetable'
print(x, maxrows, ...)
```

Arguments

`x`	an object of S3-class `probetable`.
`maxrows`	maximum number of rows to print.
`...`	further arguments that get ignored.

See Also

`print.data.frame`

Examples

```
a = as.data.frame(matrix(runif(1e6), ncol=1e3))
class(a) = c("probetable", class(a))
print(a)
print(as.matrix(a[2:3, 4:6]))
```

`toggleProbes` *Methods for getting/setting the filters on a Bimap object*

Description

These methods are part of the Bimap interface (see `?Bimap` for a quick overview of the Bimap objects and their interface).

Some of these methods are for getting or setting the filtering status on a Bimap object so that the mapping object can toggle between displaying all probes, only single probes (the defualt) or only multiply matching probes.

Other methods are for viewing or setting the filter threshold value on a InpAnnDbBimap object.

Usage

```
## Making a Bimap object that does not prefilter to remove probes that
## match multiple genes:
toggleProbes(x, value)
hasMultiProbes(x) ##T/F test for exposure of single probes
hasSingleProbes(x) ##T/F test for exposure of mulitply matched probes

## Looking at the SQL filter values for a Bimap
getBimapFilters(x)
## Setting the filter on an InpAnnDbBimap object
setInpBimapFilter(x,value)
```

Arguments

`x`	A Bimap object.
`value`	A character vector containing the new value that the Bimap should use as the filter. Or the value to toggle a probe mapping to: "all", "single", or "multiple".

Details

`toggleProbes(x)` is a methods for creating Bimaps that have an alternate filter for which probes get exposed based upon whether these probes map to multiple genes or not.

`hasMultiProbes(x)` and `hasSingleProbes(x)` are provided to give a quick test about whether or not such probes are exposed in a given mapping.

`getBimapFilters(x)` will list all the SQL filters applied to a Bimap object.

`setInpBimapFilters(x)` will allow you to pass a value as a character string which will be used as a filter. In order to be useful with the InpAnnDbBimap objects provided in the inparanoid packages, this value needs to be a to digit number written as a percentage. So for example "80 is owing to the nature of the inparanoid data set.

Value

A Bimap object of the same subtype as x for exposeAllProbes(x), maskMultiProbes(x) and maskSingleProbes(x).

A TRUE or FALSE value in the case of hasMultiProbes(x) and hasSingleProbes(x).

Author(s)

M. Carlson

See Also

Bimap, Bimap-keys, Bimap-direction, BimapFormatting, Bimap-envirAPI, nhit

Examples

```
## Make a Bimap that contains all the probes
require("hgu95av2.db")
mapWithMultiProbes <- toggleProbes(hgu95av2ENTREZID, "all")
count.mappedLkeys(hgu95av2ENTREZID)
count.mappedLkeys(mapWithMultiProbes)

## Check that it has both multiply and singly matching probes:
hasMultiProbes(mapWithMultiProbes)
hasSingleProbes(mapWithMultiProbes)

## Make it have Multi probes ONLY:
OnlyMultiProbes = toggleProbes(mapWithMultiProbes, "multiple")
hasMultiProbes(OnlyMultiProbes)
hasSingleProbes(OnlyMultiProbes)

## Convert back to a default map with only single probes exposed
OnlySingleProbes = toggleProbes(OnlyMultiProbes, "single")
hasMultiProbes(OnlySingleProbes)
hasSingleProbes(OnlySingleProbes)


## List the filters on the inparanoid mapping
# library(hom.Dm.inp.db)
# getBimapFilters(hom.Dm.inpANOGA)

## Here is how you can make a mapping with a
##different filter than the default:
# f80 = setInpBimapFilter(hom.Dm.inpANOGA, "80%")
# dim(hom.Dm.inpANOGA)
# dim(f80)
```

---

toSQLStringSet *Convert a vector to a quoted string for use as a SQL value list*

Description

Given a vector, this function returns a string with each element of the input coerced to character, quoted, and separated by ",".

Usage

```
toSQLStringSet(names)
```

Arguments

names	A vector of values to quote

Details

If names is a character vector with elements containing single quotes, these quotes will be doubled so as to escape the quote in SQL.

Value

A character vector of length one that represents the input vector as a SQL value list. Each element is single quoted and elements are comma separated.

Note

Do not use sQuote for generating SQL as that function is intended for display purposes only. In some locales, sQuote will generate fancy quotes which will break your SQL.

Author(s)

Herve Pages

Examples

```
toSQLStringSet(letters[1:4])
toSQLStringSet(c("'foo'", "ab'cd", "bar"))
```

---

unlist2 *A replacement for unlist() that does not mangle the names*

Description

unlist2 is a replacement for base::unlist() that does not mangle the names.

Usage

```
unlist2(x, recursive=TRUE, use.names=TRUE, what.names="inherited")
```

Arguments

x, recursive, use.names	See ?unlist.
what.names	"inherited" or "full".

Details

Use this function if you don't like the mangled names returned by the standard `unlist` function from the base package. Using `unlist` with annotation data is dangerous and it is highly recommended to use `unlist2` instead.

Author(s)

Herve Pages

See Also

`unlist`

Examples

```
x <- list(A=c(b=-4, 2, b=7), B=3:-1, c(a=1, a=-2), C=list(c(2:-1, d=55), e=99))
unlist(x)
unlist2(x)

library(hgu95av2.db)
egids <- c("10", "100", "1000")
egids2pbids <- mget(egids, revmap(hgu95av2ENTREZID))
egids2pbids

unlist(egids2pbids)   # 1001, 1002, 10001 and 10002 are not real
                      # Entrez ids but are the result of unlist()
                      # mangling the names!

unlist2(egids2pbids)  # much cleaner! yes the names are not unique
                      # but at least they are correct...
```

Index