Package 'metacoder'

April 11, 2026

Title Tools for Parsing, Manipulating, and Graphing Taxonomic Abundance Data

Version 0.3.9

Maintainer Zachary Foster <zacharyfoster1989@gmail.com>

Description Reads, plots, and manipulates large taxonomic data sets, like those generated from modern high-throughput sequencing, such as metabarcoding (i.e. amplification metagenomics, 16S metagenomics, etc). It provides a tree-based visualization called ``heat trees'' used to depict statistics for every taxon in a taxonomy using color and size. It also provides various functions to do common tasks in microbiome bioinformatics on data in the 'taxmap' format defined by the 'taxa' package. The 'metacoder' package is described in the publication by Foster et al. (2017) <doi:10.1371/journal.pcbi.1005404>.

Depends R (>= 3.0.2)

License GPL-2 | GPL-3

LazyData true

URL https://grunwaldlab.github.io/metacoder_documentation/

BugReports https://github.com/grunwaldlab/metacoder/issues

Imports stringr, ggplot2, igraph, grid, taxize, seqinr, RCurl, ape, stats, grDevices, utils, lazyeval, dplyr, magrittr, readr, rlang, ggfittext, vegan, cowplot, GA, Rcpp, crayon, tibble, R6

Suggests knitr, rmarkdown, testthat, BiocManager, phyloseq, phylotate, traits, biomformat, DESeq2

VignetteBuilder knitr

RoxygenNote 7.3.3

Encoding UTF-8

LinkingTo Rcpp

NeedsCompilation yes

Author Zachary Foster [aut, cre], Niklaus Grunwald [ths],

Kamil Slowikowski [ctb],
Scott Chamberlain [ctb],
Rob Gilmore [ctb]

Repository CRAN

Date/Publication 2026-04-11 06:30:02 UTC

Contents

`all_names` *Return names of data in [taxonomy()] or [taxmap()]*

Description

Return the names of data that can be used with functions in the taxa package that use [non-standard evaluation](http://adv-r.had.co.nz/Computing-on-the-language.html) (NSE), like [filter_taxa()].

```
obj$all_names(tables = TRUE, funcs = TRUE,
  others = TRUE, warn = FALSE)
all_names(obj, tables = TRUE, funcs = TRUE,
  others = TRUE, warn = FALSE)
```

Arguments

`obj`	([taxonomy()] or [taxmap()]) The object containing taxon information to be queried.
`tables`	This option only applies to [taxmap()] objects. If 'TRUE', include the names of columns of tables in 'obj$data'
`funcs`	This option only applies to [taxmap()] objects. If 'TRUE', include the names of user-definable functions in 'obj$funcs'.
`others`	This option only applies to [taxmap()] objects. If 'TRUE', include the names of data in 'obj$data' besides tables.
`builtin_funcs`	This option only applies to [taxmap()] objects. If 'TRUE', include functions like [n_supertaxa()] that provide information for each taxon.
`warn`	option only applies to [taxmap()] objects. If 'TRUE', warn if there are duplicate names. Duplicate names make it unclear what data is being referred to.

Value

'character'

See Also

Other NSE helpers: `data_used`, `get_data()`, `names_used`

Examples

```
# Get the names of all data accesible by non-standard evaluation
all_names(ex_taxmap)

# Dont include the names of automatically included functions.
all_names(ex_taxmap, builtin_funcs = FALSE)
```

ambiguous_synonyms *Get patterns for ambiguous taxa*

Description

This function stores the regex patterns for ambiguous taxa.

Usage

```
ambiguous_synonyms(
  unknown = TRUE,
  uncultured = TRUE,
  regex = TRUE,
  case_variations = FALSE
)
```

Arguments

unknown	If TRUE, include names that suggest they are placeholders for unknown taxa (e.g. "unknown ...").
uncultured	If TRUE, include names that suggest they are assigned to uncultured organisms (e.g. "uncultured ...").
regex	If TRUE, includes regex syntax to make matching things like spaces more robust.
case_variations	If TRUE, include variations of letter case.

arrange_obs *Sort user data in [taxmap()] objects*

Description

Sort rows of tables or the elements of lists/vectors in the ‘obj$data‘ list in [taxmap()] objects. Any variable name that appears in [all_names()] can be used as if it was a vector on its own. See [dplyr::arrange()] for the inspiration for this function and more information. Calling the function using the ‘obj$arrange_obs(...)‘ style edits "obj" in place, unlike most R functions. However, calling the function using the ‘arrange_obs(obj, ...)‘ imitates R’s traditional copy-on-modify semantics, so "obj" would not be changed; instead a changed version would be returned, like most R functions.

```
obj$arrange_obs(data, ...)
arrange_obs(obj, data, ...)
```

Arguments

obj	An object of type [taxmap()].
data	Dataset names, indexes, or a logical vector that indicates which datasets in ‘obj$data‘ to sort If multiple datasets are sorted at once, then they must be the same length.
...	One or more expressions (e.g. column names) to sort on.
target	DEPRECIATED. use "data" instead.

Value

An object of type [taxmap()]

See Also

Other taxmap manipulation functions: arrange_taxa(), filter_obs(), filter_taxa(), mutate_obs(), sample_frac_obs(), sample_frac_taxa(), sample_n_obs(), sample_n_taxa(), select_obs(), transmute_obs()

Examples

```
# Sort in ascending order
arrange_obs(ex_taxmap, "info", n_legs)
arrange_obs(ex_taxmap, "foods", name)

# Sort in decending order
arrange_obs(ex_taxmap, "info", desc(n_legs))

# Sort multiple datasets at once
arrange_obs(ex_taxmap, c("info", "phylopic_ids", "foods"), n_legs)
```

---

arrange_taxa *Sort the edge list of [taxmap()] objects*

Description

Sort the edge list and taxon list in [taxonomy()] or [taxmap()] objects. See [dplyr::arrange()] for the inspiration for this function and more information. Calling the function using the ‘obj$arrange_taxa(...)‘ style edits "obj" in place, unlike most R functions. However, calling the function using the ‘arrange_taxa(obj, ...)‘ imitates R’s traditional copy-on-modify semantics, so "obj" would not be changed; instead a changed version would be returned, like most R functions.

```
obj$arrange_taxa(...)
arrange_taxa(obj, ...)
```

Arguments

obj	[taxonomy()] or [taxmap()]
...	One or more expressions (e.g. column names) to sort on. Any variable name that appears in [all_names()] can be used as if it was a vector on its own.

Value

An object of type [taxonomy()] or [taxmap()]

See Also

Other taxmap manipulation functions: `arrange_obs()`, `filter_obs()`, `filter_taxa()`, `mutate_obs()`, `sample_frac_obs()`, `sample_frac_taxa()`, `sample_n_obs()`, `sample_n_taxa()`, `select_obs()`, `transmute_obs()`

Examples

```
# Sort taxa in ascending order
arrange_taxa(ex_taxmap, taxon_names)

# Sort taxa in decending order
arrange_taxa(ex_taxmap, desc(taxon_names))

# Sort using an expression. List genera first.
arrange_taxa(ex_taxmap, taxon_ranks != "genus")
```

---

as_phyloseq *Convert taxmap to phyloseq*

Description

Convert a taxmap object to a phyloseq object.

Usage

```
as_phyloseq(
  obj,
  otu_table = NULL,
  otu_id_col = "otu_id",
  sample_data = NULL,
  sample_id_col = "sample_id",
  phy_tree = NULL
)
```

Arguments

`obj`	The taxmap object.
`otu_table`	The table in ‘obj$data‘ with OTU counts. Must be one of the following: `NULL` Look for a table named "otu_table" in ‘obj$data‘ with taxon IDs, OTU IDs, and OTU counts. If it exists, use it. `character` The name of the table stored in ‘obj$data‘ with taxon IDs, OTU IDs, and OTU counts `data.frame` A table with taxon IDs, OTU IDs, and OTU counts `FALSE` Do not include an OTU table, even if "otu_table" exists in ‘obj$data‘
`otu_id_col`	The name of the column storing OTU IDs in the OTU table.
`sample_data`	A table containing sample data with sample IDs matching column names in the OTU table. Must be one of the following: `NULL` Look for a table named "sample_data" in ‘obj$data‘. If it exists, use it. `character` The name of the table stored in ‘obj$data‘ with sample IDs `data.frame` A table with sample IDs `FALSE` Do not include a sample data table, even if "sample_data" exists in ‘obj$data‘
`sample_id_col`	The name of the column storing sample IDs in the sample data table.
`phy_tree`	A phylogenetic tree of class `ape:phylo` from the `ape` package with tip labels matching OTU ids. Must be one of the following: `NULL` Look for a tree named "phy_tree" in ‘obj$data‘ with tip labels matching OTU ids. If it exists, use it. `character` The name of the tree stored in ‘obj$data‘ with tip labels matching OTU ids. `ape::phylo` A tree with tip labels matching OTU ids. `FALSE` Do not include a tree, even if "phy_tree" exists in ‘obj$data‘

Examples

```
if (requireNamespace("phyloseq", quietly = TRUE)) {
  # Parse example dataset
  library(phyloseq)
  data(GlobalPatterns)
  x <- parse_phyloseq(GlobalPatterns)

  # Convert back to a phylseq object
  as_phyloseq(x)
}
```

`branches` *Get "branch" taxa*

Description

Return the "branch" taxa for a [taxonomy()] or [taxmap()] object. A branch is anything that is not a root, stem, or leaf. Its the interior of the tree after the first split starting from the roots. Can also be used to get the branches of a subset of taxa.

```
obj$branches(subset = NULL, value = "taxon_indexes")
branches(obj, subset = NULL, value = "taxon_indexes")
```

Arguments

`obj`	The [taxonomy()] or [taxmap()] object containing taxon information to be queried.
`subset`	Taxon IDs, TRUE/FALSE vector, or taxon indexes used to subset the tree prior to determining branches. Default: All taxa in ‘obj‘ will be used. Any variable name that appears in [all_names()] can be used as if it was a vector on its own. Note that branches are determined after the filtering, so a given taxon might be a branch on the unfiltered tree, but not a branch on the filtered tree.
`value`	What data to return. This is usually the name of column in a table in ‘obj$data‘. Any result of [all_names()] can be used, but it usually only makes sense to use data that corresponds to taxa 1:1, such as [taxon_ranks()]. By default, taxon indexes are returned.

Value

‘character‘

See Also

Other taxonomy indexing functions: `internodes()`, `leaves()`, `roots()`, `stems()`, `subtaxa()`, `supertaxa()`

Examples

```
# Return indexes of branch taxa
branches(ex_taxmap)

# Return indexes for a subset of taxa
branches(ex_taxmap, subset = 2:17)
branches(ex_taxmap, subset = n_obs > 1)

# Return something besides taxon indexes
branches(ex_taxmap, value = "taxon_names")
```

`calc_diff_abund_deseq2` *Differential abundance with DESeq2*

Description

EXPERIMENTAL: This function is still being tested and developed; use with caution. Uses the `DESeq2-package` package to conduct differential abundance analysis of count data. Counts can be of OTUs/ASVs or taxa. The plotting function `heat_tree_matrix` is useful for visualizing these results. See details section below for considerations on preparing data for this analysis.

Usage

```
calc_diff_abund_deseq2(
  obj,
  data,
  cols,
  groups,
  other_cols = FALSE,
  lfc_shrinkage = c("none", "normal", "ashr"),
  ...
)
```

Arguments

`obj`	A `taxmap` object
`data`	The name of a table in `obj` that contains data for each sample in columns.
`cols`	The names/indexes of columns in `data` to use. By default, all numeric columns are used. Takes one of the following inputs: **TRUE/FALSE:** All/No columns will used. **Character vector:** The names of columns to use **Numeric vector:** The indexes of columns to use **Vector of TRUE/FALSE of length equal to the number of columns:** Use the columns corresponding to `TRUE` values.
`groups`	A vector defining how samples are grouped into "treatments". Must be the same order and length as `cols`.
`other_cols`	If `TRUE`, preserve all columns not in `cols` in the output. If `FALSE`, dont keep other columns. If a column names or indexes are supplied, only preserve those columns.
`lfc_shrinkage`	What technique to use to adjust the log fold change results for low counts. Useful for ranking and visualizing log fold changes. Must be one of the following: **'none'** No log fold change adjustments. **'normal'** The original DESeq2 shrinkage estimator

'ashr' Adaptive shrinkage estimator from the `ashr` package, using a fitted mixture of normals prior.

`...` Passed to `results` if the `lfc_shrinkage` option is "none" and to `lfcShrink` otherwise.

Details

Data should be raw read counts, not rarefied, converted to proportions, or modified with any other technique designed to correct for sample size since `DESeq2-package` is designed to be used with count data and takes into account unequal sample size when determining differential abundance. Warnings will be given if the data is not integers or all sample sizes are equal.

Value

A tibble with at least the taxon ID of the thing tested, the groups compared, and the DESeq2 results. The `log2FoldChange` values will be positive if `treatment_1` is more abundant and `treatment_2`.

See Also

Other calculations: `calc_group_mean()`, `calc_group_median()`, `calc_group_rsd()`, `calc_group_stat()`, `calc_n_samples()`, `calc_obs_props()`, `calc_prop_samples()`, `calc_taxon_abund()`, `compare_groups()`, `counts_to_presence()`, `rarefy_obs()`, `zero_low_counts()`

Examples

```
if (requireNamespace("DESeq2", quietly = TRUE)) {
  # Parse data for plotting
  x = parse_tax_data(hmp_otus, class_cols = "lineage", class_sep = ";",
                     class_key = c(tax_rank = "taxon_rank", tax_name = "taxon_name"),
                     class_regex = "^(.+)__(.+)$")

  # Get per-taxon counts
  x$data$tax_table <- calc_taxon_abund(x, data = "tax_data", cols = hmp_samples$sample_id)

  # Calculate difference between groups
  x$data$diff_table <- calc_diff_abund_deseq2(x, data = "tax_table",
                                      cols = hmp_samples$sample_id,
                                      groups = hmp_samples$body_site)

  # Plot results (might take a few minutes)
  heat_tree_matrix(x,
                   data = "diff_table",
                   node_size = n_obs,
                   node_label = taxon_names,
                   node_color = ifelse(is.na(padj) | padj > 0.05, 0, log2FoldChange),
                   node_color_range = diverging_palette(),
                   node_color_trans = "linear",
                   node_color_interval = c(-3, 3),
                   edge_color_interval = c(-3, 3),
                   node_size_axis_label = "Number of OTUs",
                   node_color_axis_label = "Log2 fold change")
```

```
}
```

`calc_group_mean` *Calculate means of groups of columns*

Description

For a given table in a `taxmap` object, split columns by a grouping factor and return row means in a table.

Usage

```
calc_group_mean(
  obj,
  data,
  groups,
  cols = NULL,
  other_cols = FALSE,
  out_names = NULL,
  dataset = NULL
)
```

Arguments

`obj`	A `taxmap` object
`data`	The name of a table in `obj$data`.
`groups`	Group multiple columns per treatment/group. This should be a vector of group IDs (e.g. character, integer) the same length as `cols` that defines which samples go in which group. When used, there will be one column in the output for each unique value in `groups`.
`cols`	The columns in `data` to use. By default, all numeric columns are used. Takes one of the following inputs: **TRUE/FALSE:** All/No columns will used. **Character vector:** The names of columns to use **Numeric vector:** The indexes of columns to use **Vector of TRUE/FALSE of length equal to the number of columns:** Use the columns corresponding to `TRUE` values.
`other_cols`	Preserve in the output non-target columns present in the input data. New columns will always be on the end. The "taxon_id" column will be preserved in the front. Takes one of the following inputs: **NULL:** No columns will be added back, not even the taxon id column. **TRUE/FALSE:** All/None of the non-target columns will be preserved. **Character vector:** The names of columns to preserve

Numeric vector: The indexes of columns to preserve

Vector of TRUE/FALSE of length equal to the number of columns: Preserve the columns corresponding to `TRUE` values.

`out_names` The names of count columns in the output. Must be the same length and order as `cols` (or `unique(groups)`, if `groups` is used).

`dataset` DEPRECIATED. use "data" instead.

Value

A tibble

See Also

Other calculations: `calc_diff_abund_deseq2()`, `calc_group_median()`, `calc_group_rsd()`, `calc_group_stat()`, `calc_n_samples()`, `calc_obs_props()`, `calc_prop_samples()`, `calc_taxon_abund()`, `compare_groups()`, `counts_to_presence()`, `rarefy_obs()`, `zero_low_counts()`

Examples

```
# Parse data for examples
x = parse_tax_data(hmp_otus, class_cols = "lineage", class_sep = ";",
                   class_key = c(tax_rank = "taxon_rank", tax_name = "taxon_name"),
                   class_regex = "^(.+)__(.+)$")

# Calculate the means for each group
calc_group_mean(x, "tax_data", hmp_samples$sex)

# Use only some columns
calc_group_mean(x, "tax_data", hmp_samples$sex[4:20],
                cols = hmp_samples$sample_id[4:20])

# Including all other columns in ouput
calc_group_mean(x, "tax_data", groups = hmp_samples$sex,
                other_cols = TRUE)

# Inlcuding specific columns in output
calc_group_mean(x, "tax_data", groups = hmp_samples$sex,
                other_cols = 2)
calc_group_mean(x, "tax_data", groups = hmp_samples$sex,
                other_cols = "otu_id")

# Rename output columns
calc_group_mean(x, "tax_data", groups = hmp_samples$sex,
               out_names = c("Women", "Men"))
```

`calc_group_median` *Calculate medians of groups of columns*

Description

For a given table in a `taxmap` object, split columns by a grouping factor and return row medians in a table.

Usage

```
calc_group_median(
  obj,
  data,
  groups,
  cols = NULL,
  other_cols = FALSE,
  out_names = NULL,
  dataset = NULL
)
```

Arguments

`obj`	A `taxmap` object
`data`	The name of a table in `obj$data`.
`groups`	Group multiple columns per treatment/group. This should be a vector of group IDs (e.g. character, integer) the same length as `cols` that defines which samples go in which group. When used, there will be one column in the output for each unique value in `groups`.
`cols`	The columns in `data` to use. By default, all numeric columns are used. Takes one of the following inputs: **TRUE/FALSE:** All/No columns will used. **Character vector:** The names of columns to use **Numeric vector:** The indexes of columns to use **Vector of TRUE/FALSE of length equal to the number of columns:** Use the columns corresponding to `TRUE` values.
`other_cols`	Preserve in the output non-target columns present in the input data. New columns will always be on the end. The "taxon_id" column will be preserved in the front. Takes one of the following inputs: **NULL:** No columns will be added back, not even the taxon id column. **TRUE/FALSE:** All/None of the non-target columns will be preserved. **Character vector:** The names of columns to preserve **Numeric vector:** The indexes of columns to preserve **Vector of TRUE/FALSE of length equal to the number of columns:** Preserve the columns corresponding to `TRUE` values.

out_names	The names of count columns in the output. Must be the same length and order as cols (or unique(groups), if groups is used).
dataset	DEPRECIATED. use "data" instead.

Value

A tibble

See Also

Other calculations: calc_diff_abund_deseq2(), calc_group_mean(), calc_group_rsd(), calc_group_stat(), calc_n_samples(), calc_obs_props(), calc_prop_samples(), calc_taxon_abund(), compare_groups(), counts_to_presence(), rarefy_obs(), zero_low_counts()

Examples

```
# Parse data for examples
x = parse_tax_data(hmp_otus, class_cols = "lineage", class_sep = ";",
                   class_key = c(tax_rank = "taxon_rank", tax_name = "taxon_name"),
                   class_regex = "^(.+)__(.+)$")

# Calculate the medians for each group
calc_group_median(x, "tax_data", hmp_samples$sex)

# Use only some columns
calc_group_median(x, "tax_data", hmp_samples$sex[4:20],
                  cols = hmp_samples$sample_id[4:20])

# Including all other columns in ouput
calc_group_median(x, "tax_data", groups = hmp_samples$sex,
                  other_cols = TRUE)

# Inlcuding specific columns in output
calc_group_median(x, "tax_data", groups = hmp_samples$sex,
                  other_cols = 2)
calc_group_median(x, "tax_data", groups = hmp_samples$sex,
                  other_cols = "otu_id")

# Rename output columns
calc_group_median(x, "tax_data", groups = hmp_samples$sex,
                  out_names = c("Women", "Men"))
```

---

calc_group_rsd *Relative standard deviations of groups of columns*

Description

For a given table in a taxmap object, split columns by a grouping factor and return the relative standard deviation for each row in a table. The relative standard deviation is the standard deviation divided by the mean of a set of numbers. It is useful for comparing the variation when magnitude of sets of number are very different.

Usage

```
calc_group_rsd(
  obj,
  data,
  groups,
  cols = NULL,
  other_cols = FALSE,
  out_names = NULL,
  dataset = NULL
)
```

Arguments

obj	A taxmap object
data	The name of a table in obj$data.
groups	Group multiple columns per treatment/group. This should be a vector of group IDs (e.g. character, integer) the same length as cols that defines which samples go in which group. When used, there will be one column in the output for each unique value in groups.
cols	The columns in data to use. By default, all numeric columns are used. Takes one of the following inputs: **TRUE/FALSE:** All/No columns will used. **Character vector:** The names of columns to use **Numeric vector:** The indexes of columns to use **Vector of TRUE/FALSE of length equal to the number of columns:** Use the columns corresponding to TRUE values.
other_cols	Preserve in the output non-target columns present in the input data. New columns will always be on the end. The "taxon_id" column will be preserved in the front. Takes one of the following inputs: **NULL:** No columns will be added back, not even the taxon id column. **TRUE/FALSE:** All/None of the non-target columns will be preserved. **Character vector:** The names of columns to preserve **Numeric vector:** The indexes of columns to preserve **Vector of TRUE/FALSE of length equal to the number of columns:** Preserve the columns corresponding to TRUE values.
out_names	The names of count columns in the output. Must be the same length and order as cols (or unique(groups), if groups is used).
dataset	DEPRECIATED. use "data" instead.

Value

A tibble

See Also

Other calculations: calc_diff_abund_deseq2(), calc_group_mean(), calc_group_median(), calc_group_stat(), calc_n_samples(), calc_obs_props(), calc_prop_samples(), calc_taxon_abund(), compare_groups(), counts_to_presence(), rarefy_obs(), zero_low_counts()

Examples

```
# Parse data for examples
x = parse_tax_data(hmp_otus, class_cols = "lineage", class_sep = ";",
                   class_key = c(tax_rank = "taxon_rank", tax_name = "taxon_name"),
                   class_regex = "^(.+)__(.+)$")

# Calculate the RSD for each group
calc_group_rsd(x, "tax_data", hmp_samples$sex)

# Use only some columns
calc_group_rsd(x, "tax_data", hmp_samples$sex[4:20],
                cols = hmp_samples$sample_id[4:20])

# Including all other columns in ouput
calc_group_rsd(x, "tax_data", groups = hmp_samples$sex,
                other_cols = TRUE)

# Inlcuding specific columns in output
calc_group_rsd(x, "tax_data", groups = hmp_samples$sex,
                other_cols = 2)
calc_group_rsd(x, "tax_data", groups = hmp_samples$sex,
                other_cols = "otu_id")

# Rename output columns
calc_group_rsd(x, "tax_data", groups = hmp_samples$sex,
               out_names = c("Women", "Men"))
```

---

calc_group_stat *Apply a function to groups of columns*

Description

For a given table in a taxmap object, apply a function to rows in groups of columns. The result of the function is used to create new columns. This is equivalent to splitting columns of a table by a factor and using apply on each group.

Usage

```
calc_group_stat(
  obj,
  data,
  func,
  groups = NULL,
```

```
  cols = NULL,
  other_cols = FALSE,
  out_names = NULL,
  dataset = NULL
)
```

Arguments

obj	A taxmap object
data	The name of a table in obj$data.
func	The function to apply. It should take a vector and return a single value. For example, max or mean could be used.
groups	Group multiple columns per treatment/group. This should be a vector of group IDs (e.g. character, integer) the same length as cols that defines which samples go in which group. When used, there will be one column in the output for each unique value in groups.
cols	The columns in data to use. By default, all numeric columns are used. Takes one of the following inputs: **TRUE/FALSE:** All/No columns will used. **Character vector:** The names of columns to use **Numeric vector:** The indexes of columns to use **Vector of TRUE/FALSE of length equal to the number of columns:** Use the columns corresponding to TRUE values.
other_cols	Preserve in the output non-target columns present in the input data. New columns will always be on the end. The "taxon_id" column will be preserved in the front. Takes one of the following inputs: **NULL:** No columns will be added back, not even the taxon id column. **TRUE/FALSE:** All/None of the non-target columns will be preserved. **Character vector:** The names of columns to preserve **Numeric vector:** The indexes of columns to preserve **Vector of TRUE/FALSE of length equal to the number of columns:** Preserve the columns corresponding to TRUE values.
out_names	The names of count columns in the output. Must be the same length and order as cols (or unique(groups), if groups is used).
dataset	DEPRECIATED. use "data" instead.

Value

A tibble

See Also

Other calculations: calc_diff_abund_deseq2(), calc_group_mean(), calc_group_median(), calc_group_rsd(), calc_n_samples(), calc_obs_props(), calc_prop_samples(), calc_taxon_abund(), compare_groups(), counts_to_presence(), rarefy_obs(), zero_low_counts()

Examples

```
# Parse data for examples
x = parse_tax_data(hmp_otus, class_cols = "lineage", class_sep = ";",
                   class_key = c(tax_rank = "taxon_rank", tax_name = "taxon_name"),
                   class_regex = "^(.+)__(.+)$")

# Apply a function to every value without grouping
calc_group_stat(x, "tax_data", function(v) v > 3)

# Calculate the means for each group
calc_group_stat(x, "tax_data", mean, groups = hmp_samples$sex)

# Calculate the variation for each group
calc_group_stat(x, "tax_data", sd, groups = hmp_samples$body_site)

# Different ways to use only some columns
calc_group_stat(x, "tax_data", function(v) v > 3,
                cols = c("700035949", "700097855", "700100489"))
calc_group_stat(x, "tax_data", function(v) v > 3,
                cols = 4:6)
calc_group_stat(x, "tax_data", function(v) v > 3,
                cols = startsWith(colnames(x$data$tax_data), "70001"))

# Including all other columns in ouput
calc_group_stat(x, "tax_data", mean, groups = hmp_samples$sex,
                other_cols = TRUE)

# Inlcuding specific columns in output
calc_group_stat(x, "tax_data", mean, groups = hmp_samples$sex,
                other_cols = 2)
calc_group_stat(x, "tax_data", mean, groups = hmp_samples$sex,
                other_cols = "otu_id")

# Rename output columns
calc_group_stat(x, "tax_data", mean, groups = hmp_samples$sex,
               out_names = c("Women", "Men"))
```

`calc_n_samples` *Count the number of samples*

Description

For a given table in a `taxmap` object, count the number of samples (i.e. columns) with greater than a minimum value.

Usage

```
calc_n_samples(
  obj,
```

```
  data,
  cols = NULL,
  groups = "n_samples",
  other_cols = FALSE,
  out_names = NULL,
  drop = FALSE,
  more_than = 0,
  dataset = NULL
)
```

Arguments

obj	A taxmap object
data	The name of a table in obj$data.
cols	The columns in data to use. By default, all numeric columns are used. Takes one of the following inputs: **TRUE/FALSE:** All/No columns will used. **Character vector:** The names of columns to use **Numeric vector:** The indexes of columns to use **Vector of TRUE/FALSE of length equal to the number of columns:** Use the columns corresponding to TRUE values.
groups	Group multiple columns per treatment/group. This should be a vector of group IDs (e.g. character, integer) the same length as cols that defines which samples go in which group. When used, there will be one column in the output for each unique value in groups.
other_cols	Preserve in the output non-target columns present in the input data. New columns will always be on the end. The "taxon_id" column will be preserved in the front. Takes one of the following inputs: **NULL:** No columns will be added back, not even the taxon id column. **TRUE/FALSE:** All/None of the non-target columns will be preserved. **Character vector:** The names of columns to preserve **Numeric vector:** The indexes of columns to preserve **Vector of TRUE/FALSE of length equal to the number of columns:** Preserve the columns corresponding to TRUE values.
out_names	The names of count columns in the output. Must be the same length and order as cols (or unique(groups), if groups is used).
drop	If groups is not used, return a vector of the results instead of a table with one column.
more_than	A sample must have greater than this value for it to be counted as present.
dataset	DEPRECIATED. use "data" instead.

Value

A tibble

See Also

Other calculations: calc_diff_abund_deseq2(), calc_group_mean(), calc_group_median(), calc_group_rsd(), calc_group_stat(), calc_obs_props(), calc_prop_samples(), calc_taxon_abund(), compare_groups(), counts_to_presence(), rarefy_obs(), zero_low_counts()

Examples

```
# Parse data for example
x = parse_tax_data(hmp_otus, class_cols = "lineage", class_sep = ";",
                   class_key = c(tax_rank = "taxon_rank", tax_name = "taxon_name"),
                   class_regex = "^(.+)__(.+)$")

# Count samples with at least one read
calc_n_samples(x, data = "tax_data")

# Count samples with at least 5 reads
calc_n_samples(x, data = "tax_data", more_than = 5)

# Return a vector instead of a table
calc_n_samples(x, data = "tax_data", drop = TRUE)

# Only use some columns
calc_n_samples(x, data = "tax_data", cols = hmp_samples$sample_id[1:5])

# Return a count for each treatment
calc_n_samples(x, data = "tax_data", groups = hmp_samples$body_site)

# Rename output columns
calc_n_samples(x, data = "tax_data", groups = hmp_samples$body_site,
               out_names = c("A", "B", "C", "D", "E"))

# Preserve other columns from input
calc_n_samples(x, data = "tax_data", other_cols = TRUE)
calc_n_samples(x, data = "tax_data", other_cols = 2)
calc_n_samples(x, data = "tax_data", other_cols = "otu_id")
```

---

calc_obs_props *Calculate proportions from observation counts*

Description

For a given table in a taxmap object, convert one or more columns containing counts to proportions. This is meant to be used with counts associated with observations (e.g. OTUs), as opposed to counts that have already been summed per taxon.

Usage

```
calc_obs_props(
  obj,
  data,
  cols = NULL,
  groups = NULL,
  other_cols = FALSE,
  out_names = NULL,
  dataset = NULL
)
```

Arguments

obj	A taxmap object
data	The name of a table in obj$data.
cols	The columns in data to use. By default, all numeric columns are used. Takes one of the following inputs: **TRUE/FALSE:** All/No columns will used. **Character vector:** The names of columns to use **Numeric vector:** The indexes of columns to use **Vector of TRUE/FALSE of length equal to the number of columns:** Use the columns corresponding to TRUE values.
groups	Group multiple columns per treatment/group. This should be a vector of group IDs (e.g. character, integer) the same length as cols that defines which samples go in which group. When used, there will be one column in the output for each unique value in groups.
other_cols	Preserve in the output non-target columns present in the input data. New columns will always be on the end. The "taxon_id" column will be preserved in the front. Takes one of the following inputs: **NULL:** No columns will be added back, not even the taxon id column. **TRUE/FALSE:** All/None of the non-target columns will be preserved. **Character vector:** The names of columns to preserve **Numeric vector:** The indexes of columns to preserve **Vector of TRUE/FALSE of length equal to the number of columns:** Preserve the columns corresponding to TRUE values.
out_names	The names of count columns in the output. Must be the same length and order as cols (or unique(groups), if groups is used).
dataset	DEPRECIATED. use "data" instead.

Value

A tibble

See Also

Other calculations: calc_diff_abund_deseq2(), calc_group_mean(), calc_group_median(), calc_group_rsd(), calc_group_stat(), calc_n_samples(), calc_prop_samples(), calc_taxon_abund(), compare_groups(), counts_to_presence(), rarefy_obs(), zero_low_counts()

Examples

```
# Parse data for examples
x = parse_tax_data(hmp_otus, class_cols = "lineage", class_sep = ";",
                   class_key = c(tax_rank = "taxon_rank", tax_name = "taxon_name"),
                   class_regex = "^(.+)__(.+)$")

# Calculate proportions for all numeric columns
calc_obs_props(x, "tax_data")

# Calculate proportions for a subset of columns
calc_obs_props(x, "tax_data", cols = c("700035949", "700097855", "700100489"))
calc_obs_props(x, "tax_data", cols = 4:6)
calc_obs_props(x, "tax_data", cols = startsWith(colnames(x$data$tax_data), "70001"))

# Including all other columns in ouput
calc_obs_props(x, "tax_data", other_cols = TRUE)

# Inlcuding specific columns in output
calc_obs_props(x, "tax_data", cols = c("700035949", "700097855", "700100489"),
               other_cols = 2:3)

# Rename output columns
calc_obs_props(x, "tax_data", cols = c("700035949", "700097855", "700100489"),
               out_names = c("a", "b", "c"))

# Get proportions for groups of samples
calc_obs_props(x, "tax_data", groups = hmp_samples$sex)
calc_obs_props(x, "tax_data", groups = hmp_samples$sex,
               out_names = c("Women", "Men"))
```

---

calc_prop_samples *Calculate the proportion of samples*

Description

For a given table in a taxmap object, calculate the proportion of samples (i.e. columns) with greater than a minimum value.

Usage

```
calc_prop_samples(
  obj,
  data,
```

```
  cols = NULL,
  groups = "prop_samples",
  other_cols = FALSE,
  out_names = NULL,
  drop = FALSE,
  more_than = 0,
  dataset = NULL
)
```

Arguments

obj	A taxmap object
data	The name of a table in obj$data.
cols	The columns in data to use. By default, all numeric columns are used. Takes one of the following inputs: **TRUE/FALSE:** All/No columns will used. **Character vector:** The names of columns to use **Numeric vector:** The indexes of columns to use **Vector of TRUE/FALSE of length equal to the number of columns:** Use the columns corresponding to TRUE values.
groups	Group multiple columns per treatment/group. This should be a vector of group IDs (e.g. character, integer) the same length as cols that defines which samples go in which group. When used, there will be one column in the output for each unique value in groups.
other_cols	Preserve in the output non-target columns present in the input data. New columns will always be on the end. The "taxon_id" column will be preserved in the front. Takes one of the following inputs: **NULL:** No columns will be added back, not even the taxon id column. **TRUE/FALSE:** All/None of the non-target columns will be preserved. **Character vector:** The names of columns to preserve **Numeric vector:** The indexes of columns to preserve **Vector of TRUE/FALSE of length equal to the number of columns:** Preserve the columns corresponding to TRUE values.
out_names	The names of count columns in the output. Must be the same length and order as cols (or unique(groups), if groups is used).
drop	If groups is not used, return a vector of the results instead of a table with one column.
more_than	A sample must have greater than this value for it to be counted as present.
dataset	DEPRECIATED. use "data" instead.

Value

A tibble

See Also

Other calculations: calc_diff_abund_deseq2(), calc_group_mean(), calc_group_median(), calc_group_rsd(), calc_group_stat(), calc_n_samples(), calc_obs_props(), calc_taxon_abund(), compare_groups(), counts_to_presence(), rarefy_obs(), zero_low_counts()

Examples

```
# Parse data for example
x = parse_tax_data(hmp_otus, class_cols = "lineage", class_sep = ";",
                   class_key = c(tax_rank = "taxon_rank", tax_name = "taxon_name"),
                   class_regex = "^(.+)__(.+)$")

# Count samples with at least one read
calc_prop_samples(x, data = "tax_data")

# Count samples with at least 5 reads
calc_prop_samples(x, data = "tax_data", more_than = 5)

# Return a vector instead of a table
calc_prop_samples(x, data = "tax_data", drop = TRUE)

# Only use some columns
calc_prop_samples(x, data = "tax_data", cols = hmp_samples$sample_id[1:5])

# Return a count for each treatment
calc_prop_samples(x, data = "tax_data", groups = hmp_samples$body_site)

# Rename output columns
calc_prop_samples(x, data = "tax_data", groups = hmp_samples$body_site,
                out_names = c("A", "B", "C", "D", "E"))

# Preserve other columns from input
calc_prop_samples(x, data = "tax_data", other_cols = TRUE)
calc_prop_samples(x, data = "tax_data", other_cols = 2)
calc_prop_samples(x, data = "tax_data", other_cols = "otu_id")
```

---

classifications *Get classifications of taxa*

Description

Get character vector classifications of taxa in an object of type [taxonomy()] or [taxmap()] composed of data associated with taxa. Each classification is constructed by concatenating the data of the given taxon and all of its supertaxa.

```
obj$classifications(value = "taxon_names", sep = ";")
classifications(obj, value = "taxon_names", sep = ";")
```

Arguments

obj	([taxonomy()] or [taxmap()])
value	What data to return. Any result of 'all_names(obj)' can be used, but it usually only makes sense to data that corresponds to taxa 1:1, such as [taxon_ranks()]. By default, taxon indexes are returned.
sep	('character' of length 1) The character(s) to place between taxon IDs

Value

'character'

See Also

Other taxonomy data functions: id_classifications(), is_branch(), is_internode(), is_leaf(), is_root(), is_stem(), map_data(), map_data_(), n_leaves(), n_leaves_1(), n_subtaxa(), n_subtaxa_1(), n_supertaxa(), n_supertaxa_1(), taxon_ids(), taxon_indexes(), taxon_names(), taxon_ranks()

Examples

```
# Defualt settings returns taxon names separated by ;
classifications(ex_taxmap)

# Other values can be returned besides taxon names
classifications(ex_taxmap, value = "taxon_ids")

# The separator can also be changed
classifications(ex_taxmap, value = "taxon_ranks", sep = "||")
```

---

compare_groups *Compare groups of samples*

Description

Apply a function to compare data, usually abundance, from pairs of treatments/groups. By default, every pairwise combination of treatments are compared. A custom function can be supplied to perform the comparison. The plotting function heat_tree_matrix is useful for visualizing these results.

Usage

```
compare_groups(
  obj,
  data,
  cols,
  groups,
```

```
  func = NULL,
  combinations = NULL,
  other_cols = FALSE,
  dataset = NULL
)
```

Arguments

obj	A taxmap object
data	The name of a table in obj that contains data for each sample in columns.
cols	The names/indexes of columns in data to use. By default, all numeric columns are used. Takes one of the following inputs: **TRUE/FALSE:** All/No columns will used. **Character vector:** The names of columns to use **Numeric vector:** The indexes of columns to use **Vector of TRUE/FALSE of length equal to the number of columns:** Use the columns corresponding to TRUE values.
groups	A vector defining how samples are grouped into "treatments". Must be the same order and length as cols.

func	The function to apply for each comparison. For each row in data, for each combination of groups, this function will receive the data for each treatment, passed as two vectors. Therefore the function must take at least 2 arguments corresponding to the two groups compared. The function should return a vector or list of results of a fixed length. If named, the names will be used in the output. The names should be consistent as well. A simple example is function(x, y) mean(x) - mean(y). By default, the following function is used:
	`function(abund_1, abund_2) {`
	`  log_ratio <- log2(median(abund_1) / median(abund_2))`
	`  if (is.nan(log_ratio)) {`
	`  log_ratio <- 0`
	`  }`
	`  list(log2_median_ratio = log_ratio,`
	`  median_diff = median(abund_1) - median(abund_2),`
	`  mean_diff = mean(abund_1) - mean(abund_2),`
	`  wilcox_p_value = wilcox.test(abund_1, abund_2)$p.value)`
	`}`

combinations	Which combinations of groups to use. Must be a list of vectors, each containing the names of 2 groups to compare. By default, all pairwise combinations of groups are compared.
other_cols	If TRUE, preserve all columns not in cols in the output. If FALSE, dont keep other columns. If a column names or indexes are supplied, only preserve those columns.
dataset	DEPRECIATED. use "data" instead.

Value

A tibble

See Also

Other calculations: calc_diff_abund_deseq2(), calc_group_mean(), calc_group_median(), calc_group_rsd(), calc_group_stat(), calc_n_samples(), calc_obs_props(), calc_prop_samples(), calc_taxon_abund(), counts_to_presence(), rarefy_obs(), zero_low_counts()

Examples

```
# Parse data for plotting
x = parse_tax_data(hmp_otus, class_cols = "lineage", class_sep = ";",
                   class_key = c(tax_rank = "taxon_rank", tax_name = "taxon_name"),
                   class_regex = "^(.+)__(.+)$")

# Convert counts to proportions
x$data$otu_table <- calc_obs_props(x, data = "tax_data", cols = hmp_samples$sample_id)

# Get per-taxon counts
x$data$tax_table <- calc_taxon_abund(x, data = "otu_table", cols = hmp_samples$sample_id)

# Calculate difference between groups
x$data$diff_table <- compare_groups(x, data = "tax_table",
                                    cols = hmp_samples$sample_id,
                                    groups = hmp_samples$body_site)

# Plot results (might take a few minutes)
heat_tree_matrix(x,
                 data = "diff_table",
                 node_size = n_obs,
                 node_label = taxon_names,
                 node_color = log2_median_ratio,
                 node_color_range = diverging_palette(),
                 node_color_trans = "linear",
                 node_color_interval = c(-3, 3),
                 edge_color_interval = c(-3, 3),
                 node_size_axis_label = "Number of OTUs",
                 node_color_axis_label = "Log2 ratio median proportions")

# How to get results for only some pairs of groups
compare_groups(x, data = "tax_table",
               cols = hmp_samples$sample_id,
               groups = hmp_samples$body_site,
               combinations = list(c('Nose', 'Saliva'),
                                   c('Skin', 'Throat')))
```

`complement` *Find complement of sequences*

Description

Find the complement of one or more sequences stored as a character vector. This is a wrapper for `comp` for character vectors instead of lists of character vectors with one value per letter. IUPAC ambiguity code are handled and the upper/lower case is preserved.

Usage

```
complement(seqs)
```

Arguments

seqs	A character vector with one element per sequence.

See Also

Other sequence transformations: `rev_comp()`, `reverse()`

Examples

```
complement(c("aagtgGGTGaa", "AAGTGGT"))
```

---

counts_to_presence *Apply a function to groups of columns*

Description

For a given table in a `taxmap` object, apply a function to rows in groups of columns. The result of the function is used to create new columns. This is equivalent to splitting columns of a table by a factor and using `apply` on each group.

Usage

```
counts_to_presence(
  obj,
  data,
  threshold = 0,
  groups = NULL,
  cols = NULL,
  other_cols = FALSE,
  out_names = NULL,
  dataset = NULL
)
```

Arguments

obj	A taxmap object
data	The name of a table in obj$data.
threshold	The value a number must be greater than to count as present. By, default, anything above 0 is considered present.
groups	Group multiple columns per treatment/group. This should be a vector of group IDs (e.g. character, integer) the same length as cols that defines which samples go in which group. When used, there will be one column in the output for each unique value in groups.
cols	The columns in data to use. By default, all numeric columns are used. Takes one of the following inputs: **TRUE/FALSE:** All/No columns will used. **Character vector:** The names of columns to use **Numeric vector:** The indexes of columns to use **Vector of TRUE/FALSE of length equal to the number of columns:** Use the columns corresponding to TRUE values.
other_cols	Preserve in the output non-target columns present in the input data. New columns will always be on the end. The "taxon_id" column will be preserved in the front. Takes one of the following inputs: **NULL:** No columns will be added back, not even the taxon id column. **TRUE/FALSE:** All/None of the non-target columns will be preserved. **Character vector:** The names of columns to preserve **Numeric vector:** The indexes of columns to preserve **Vector of TRUE/FALSE of length equal to the number of columns:** Preserve the columns corresponding to TRUE values.
out_names	The names of count columns in the output. Must be the same length and order as cols (or unique(groups), if groups is used).
dataset	DEPRECIATED. use "data" instead.

Value

A tibble

See Also

Other calculations: calc_diff_abund_deseq2(), calc_group_mean(), calc_group_median(), calc_group_rsd(), calc_group_stat(), calc_n_samples(), calc_obs_props(), calc_prop_samples(), calc_taxon_abund(), compare_groups(), rarefy_obs(), zero_low_counts()

Examples

```
# Parse data for examples
x = parse_tax_data(hmp_otus, class_cols = "lineage", class_sep = ";",
                   class_key = c(tax_rank = "taxon_rank", tax_name = "taxon_name"),
                   class_regex = "^(.+)__(.+)$")
```

```
# Convert count to presence/absence
counts_to_presence(x, "tax_data")

# Check if there are any reads in each group of samples
counts_to_presence(x, "tax_data", groups = hmp_samples$body_site)
```

`database_list` *Database list*

Description

The list of known databases. Not currently used much, but will be when we add more check for taxon IDs and taxon ranks from particular databases.

Usage

```
database_list
```

Format

An object of class `list` of length 8.

Details

List of databases with pre-filled details, where each has the format:

- url: A base URL for the database source.
- description: Description of the database source.
- id regex: identifier regex.

See Also

[taxon_database]

Examples

```
database_list
database_list$ncbi
database_list$ncbi$name
database_list$ncbi$description
database_list$ncbi$url
```

diverging_palette *The default diverging color palette*

Description

Returns the default color palette for diverging data

Usage

```
diverging_palette()
```

Value

`character` of hex color codes

Examples

```
diverging_palette()
```

extract_tax_data *Extracts taxonomy info from vectors with regex*

Description

Convert taxonomic information in a character vector into a [taxmap()] object. The location and identity of important information in the input is specified using a [regular expression](https://en.wikipedia.org/wiki/Regular_expre with capture groups and a corresponding key. An object of type [taxmap()] is returned containing the specified information. See the 'key' option for accepted sources of taxonomic information.

Usage

```
extract_tax_data(
  tax_data,
  key,
  regex,
  class_key = "taxon_name",
  class_regex = "(.*)",
  class_sep = NULL,
  sep_is_regex = FALSE,
  class_rev = FALSE,
  database = "ncbi",
  include_match = FALSE,
  include_tax_data = TRUE
)
```

Arguments

tax_data	A vector from which to extract taxonomy information.
key	(‘character‘) The identity of the capturing groups defined using ‘regex‘. The length of ‘key‘ must be equal to the number of capturing groups specified in ‘regex‘. Any names added to the terms will be used as column names in the output. Only ‘"info"‘ can be used multiple times. Each term must be one of those described below: * ‘taxon_id‘: A unique numeric id for a taxon for a particular ‘database‘ (e.g. ncbi accession number). Requires an internet connection. * ‘taxon_name‘: The name of a taxon (e.g. "Mammalia" or "Homo sapiens"). Not necessarily unique, but interpretable by a particular ‘database‘. Requires an internet connection. * ‘fuzzy_name‘: The name of a taxon, but check for misspellings first. Only use if you think there are misspellings. Using ‘"taxon_name"‘ is faster. * ‘class‘: A list of taxon information that constitutes the full taxonomic classification (e.g. "K_Mammalia;P_Carnivora;C_Felidae"). Individual taxa are separated by the ‘class_sep‘ argument and the information is parsed by the ‘class_regex‘ and ‘class_key‘ arguments. * ‘seq_id‘: Sequence ID for a particular database that is associated with a taxonomic classification. Currently only works with the "ncbi" database. * ‘info‘: Arbitrary taxon info you want included in the output. Can be used more than once.
regex	(‘character‘ of length 1) A regular expression with capturing groups indicating the locations of relevant information. The identity of the information must be specified using the ‘key‘ argument.
class_key	(‘character‘ of length 1) The identity of the capturing groups defined using ‘class_regex‘. The length of ‘class_key‘ must be equal to the number of capturing groups specified in ‘class_regex‘. Any names added to the terms will be used as column names in the output. Only ‘"info"‘ can be used multiple times. Each term must be one of those described below: * ‘taxon_name‘: The name of a taxon. Not necessarily unique. * ‘taxon_rank‘: The rank of the taxon. This will be used to add rank info into the output object that can be accessed by ‘out$taxon_ranks()‘. * ‘info‘: Arbitrary taxon info you want included in the output. Can be used more than once.
class_regex	(‘character‘ of length 1) A regular expression with capturing groups indicating the locations of data for each taxon in the ‘class‘ term in the ‘key‘ argument. The identity of the information must be specified using the ‘class_key‘ argument. The ‘class_sep‘ option can be used to split the classification into data for each taxon before matching. If ‘class_sep‘ is ‘NULL‘, each match of ‘class_regex‘ defines a taxon in the classification.
class_sep	(‘character‘ of length 1) Used with the ‘class‘ term in the ‘key‘ argument. The character(s) used to separate individual taxa within a classification. After the string defined by the ‘class‘ capture group in ‘regex‘ is split by ‘class_sep‘, its capture groups are extracted by ‘class_regex‘ and defined by ‘class_key‘. If ‘NULL‘, every match of ‘class_regex‘ is used instead with first splitting by ‘class_sep‘.
sep_is_regex	(‘TRUE‘/‘FALSE‘) Whether or not ‘class_sep‘ should be used as a [regular expression](https://en.wikipedia.org/wiki/Regular_expression).

`class_rev` (‘logical‘ of length 1) Used with the ‘class‘ term in the ‘key‘ argument. If ‘TRUE‘, the order of taxon data in a classification is reversed to be specific to broad.

`database` (‘character‘ of length 1) The name of the database that patterns given in ‘parser‘ will apply to. Valid databases include "ncbi", "itis", "eol", "col", "tropicos", "nbn", and "none". ‘"none"‘ will cause no database to be queried; use this if you want to not use the internet. NOTE: Only ‘"ncbi"‘ has been tested extensively so far.

`include_match` (‘logical‘ of length 1) If ‘TRUE‘, include the part of the input matched by ‘regex‘ in the output object.

`include_tax_data` (‘TRUE‘/‘FALSE‘) Whether or not to include ‘tax_data‘ as a dataset.

Value

Returns an object of type [taxmap()]

Failed Downloads

If you have invalid inputs or a download fails for another reason, then there will be a "unknown" taxon ID as a placeholder and failed inputs will be assigned to this ID. You can remove these using [filter_taxa()] like so: ‘filter_taxa(result, taxon_ids != "unknown")‘. Add ‘drop_obs = FALSE‘ if you want the input data, but want to remove the taxon.

See Also

Other parsers: lookup_tax_data(), parse_dada2(), parse_edge_list(), parse_greengenes(), parse_mothur_tax_summary(), parse_mothur_taxonomy(), parse_newick(), parse_phylo(), parse_phyloseq(), parse_qiime_biom(), parse_rdp(), parse_silva_fasta(), parse_tax_data(), parse_ubiome(), parse_unite_general()

Examples

```
# For demonstration purposes, the following example dataset has all the
# types of data that can be used, but any one of them alone would work.
raw_data <- c(
">id:AB548412-tid:9689-Panthera leo-tax:K_Mammalia;P_Carnivora;C_Felidae;G_Panthera;S_leo",
">id:FJ358423-tid:9694-Panthera tigris-tax:K_Mammalia;P_Carnivora;C_Felidae;G_Panthera;S_tigris",
">id:DQ334818-tid:9643-Ursus americanus-tax:K_Mammalia;P_Carnivora;C_Felidae;G_Ursus;S_americanus"
)

# Build a taxmap object from classifications
extract_tax_data(raw_data,
                 key = c(my_seq = "info", my_tid = "info", org = "info", tax = "class"),
                  regex = "^>id:(.+)-tid:(.+)-(.+)-tax:(.+)$",
                  class_sep = ";", class_regex = "^(.+)_(.+)$",
                  class_key = c(my_rank = "info", tax_name = "taxon_name"))

# Build a taxmap object from taxon ids
```

```
# Note: this requires an internet connection
extract_tax_data(raw_data,
              key = c(my_seq = "info", my_tid = "taxon_id", org = "info", tax = "info"),
                 regex = "^>id:(.+)-tid:(.+)-(.+)-tax:(.+)$")

# Build a taxmap object from ncbi sequence accession numbers
# Note: this requires an internet connection
extract_tax_data(raw_data,
               key = c(my_seq = "seq_id", my_tid = "info", org = "info", tax = "info"),
                 regex = "^>id:(.+)-tid:(.+)-(.+)-tax:(.+)$")

# Build a taxmap object from taxon names
# Note: this requires an internet connection
extract_tax_data(raw_data,
             key = c(my_seq = "info", my_tid = "info", org = "taxon_name", tax = "info"),
                 regex = "^>id:(.+)-tid:(.+)-(.+)-tax:(.+)$")
```

---

ex_hierarchies *An example hierarchies object*

Description

An example hierarchies object built from the ground up.

Format

A [hierarchies()] object.

Source

Created from the example code in the [hierarchies()] documentation.

See Also

Other taxa-datasets: ex_hierarchy1, ex_hierarchy2, ex_hierarchy3, ex_taxmap

---

ex_hierarchy1 *An example Hierarchy object*

Description

An example Hierarchy object built from the ground up.

Format

A [hierarchy()] object with

- name: Poaceae / rank: family / id: 4479
- name: Poa / rank: genus / id: 4544
- name: Poa annua / rank: species / id: 93036

Based on NCBI taxonomic classification

Source

Created from the example code in the [hierarchy()] documentation.

See Also

Other taxa-datasets: ex_hierarchies, ex_hierarchy2, ex_hierarchy3, ex_taxmap

---

ex_hierarchy2 *An example Hierarchy object*

Description

An example Hierarchy object built from the ground up.

Format

A [hierarchy()] object with

- name: Felidae / rank: family / id: 9681
- name: Puma / rank: genus / id: 146712
- name: Puma concolor / rank: species / id: 9696

Based on NCBI taxonomic classification

Source

Created from the example code in the [hierarchy()] documentation.

See Also

Other taxa-datasets: ex_hierarchies, ex_hierarchy1, ex_hierarchy3, ex_taxmap

ex_hierarchy3 *An example Hierarchy object*

Description

An example Hierarchy object built from the ground up.

Format

A [hierarchy()] object with

- name: Chordata / rank: phylum / id: 158852
- name: Vertebrata / rank: subphylum / id: 331030
- name: Teleostei / rank: class / id: 161105
- name: Salmonidae / rank: family / id: 161931
- name: Salmo / rank: genus / id: 161994
- name: Salmo salar / rank: species / id: 161996

Based on ITIS taxonomic classification

Source

Created from the example code in the [hierarchy()] documentation.

See Also

Other taxa-datasets: ex_hierarchies, ex_hierarchy1, ex_hierarchy2, ex_taxmap

ex_taxmap *An example taxmap object*

Description

An example taxmap object built from the ground up. Typically, data stored in taxmap would be parsed from an input file, but this data set is just for demonstration purposes.

Format

A [taxmap()] object.

Source

Created from the example code in the [taxmap()] documentation.

See Also

Other taxa-datasets: ex_hierarchies, ex_hierarchy1, ex_hierarchy2, ex_hierarchy3

`filtering-helpers` *Taxonomic filtering helpers*

Description

Taxonomic filtering helpers

Usage

```
ranks(...)

nms(...)

ids(...)
```

Arguments

`...`	quoted rank names, taxonomic names, taxonomic ids, or any of those with supported operators (See **Supported Relational Operators** below)

How do these functions work?

Each function assigns some metadata so we can more easily process your query downstream. In addition, we check for whether you've used any relational operators and pull those out to make downstream processing easier

The goal of these functions is to make it easy to combine queries based on each of rank names, taxonomic names, and taxonomic ids.

These are designed to be used inside of [pop()], [pick()], [span()]. Inside of those functions, we figure out what rank names you want to filter on, then check against a reference dataset ([ranks_ref]) to allow ordered queries like *I want all taxa between Class and Genus*. If you provide rank names, we just use those, then do the filtering you requested. If you provide taxonomic names or ids we figure out what rank names you are referring to, then we can proceed as in the previous sentence.

Supported Relational Operators

- '>' all items above rank of x
- '>=' all items above rank of x, inclusive
- '<' all items below rank of x
- '<=' all items below rank of x, inclusive

ranks

Ranks can be any character string in the set of acceptable rank names.

nms

‘nms‘ is named to avoid using ‘names‘ which would collide with the fxn [base::names()] in Base R. Can pass in any character taxonomic names.

ids

Ids are any alphanumeric taxonomic identifier. Some database providers use all digits, but some use a combination of digits and characters.

Note

NSE is not supported at the moment, but may be in the future

Examples

```
ranks("genus")
ranks("order", "genus")
ranks("> genus")

nms("Poaceae")
nms("Poaceae", "Poa")
nms("< Poaceae")

ids(4544)
ids(4544, 4479)
ids("< 4479")
```

---

filter_ambiguous_taxa *Filter ambiguous taxon names*

Description

Filter out taxa with ambiguous names, such as "unknown" or "uncultured". NOTE: some parameters of this function are passed to `filter_taxa` with the "invert" option set to `TRUE`. Works the same way as `filter_taxa` for the most part.

Usage

```
filter_ambiguous_taxa(
  obj,
  unknown = TRUE,
  uncultured = TRUE,
  name_regex = ".",
  ignore_case = TRUE,
  subtaxa = FALSE,
  drop_obs = TRUE,
  reassign_obs = TRUE,
  reassign_taxa = TRUE
)
```

Arguments

obj	A taxmap object
unknown	If TRUE, Remove taxa with names the suggest they are placeholders for unknown taxa (e.g. "unknown ...").
uncultured	If TRUE, Remove taxa with names the suggest they are assigned to uncultured organisms (e.g. "uncultured ...").
name_regex	The regex code to match a valid character in a taxon name. For example, "[a-z]" would mean taxon names can only be lower case letters.
ignore_case	If TRUE, dont consider the case of the text when determining a match.
subtaxa	(‘logical‘ or ‘numeric‘ of length 1) If ‘TRUE‘, include subtaxa of taxa passing the filter. Positive numbers indicate the number of ranks below the target taxa to return. ‘0‘ is equivalent to ‘FALSE‘. Negative numbers are equivalent to ‘TRUE‘.
drop_obs	(‘logical‘) This option only applies to [taxmap()] objects. If ‘FALSE‘, include observations (i.e. user-defined data in ‘obj$data‘) even if the taxon they are assigned to is filtered out. Observations assigned to removed taxa will be assigned to NA. This option can be either simply ‘TRUE‘/‘FALSE‘, meaning that all data sets will be treated the same, or a logical vector can be supplied with names corresponding one or more data sets in ‘obj$data‘. For example, ‘c(abundance = FALSE, stats = TRUE)‘ would include observations whose taxon was filtered out in ‘obj$data$abundance‘, but not in ‘obj$data$stats‘. See the ‘reassign_obs‘ option below for further complications.
reassign_obs	(‘logical‘ of length 1) This option only applies to [taxmap()] objects. If ‘TRUE‘, observations (i.e. user-defined data in ‘obj$data‘) assigned to removed taxa will be reassigned to the closest supertaxon that passed the filter. If there are no supertaxa of such an observation that passed the filter, they will be filtered out if ‘drop_obs‘ is ‘TRUE‘. This option can be either simply ‘TRUE‘/‘FALSE‘, meaning that all data sets will be treated the same, or a logical vector can be supplied with names corresponding one or more data sets in ‘obj$data‘. For example, ‘c(abundance = TRUE, stats = FALSE)‘ would reassign observations in ‘obj$data$abundance‘, but not in ‘obj$data$stats‘.
reassign_taxa	(‘logical‘ of length 1) If ‘TRUE‘, subtaxa of removed taxa will be reassigned to the closest supertaxon that passed the filter. This is useful for removing intermediate levels of a taxonomy.

Details

If you encounter a taxon name that represents an ambiguous taxon that is not filtered out by this function, let us know and we will add it.

Value

A taxmap object

Examples

```
obj <- parse_tax_data(c("Plantae;Solanaceae;Solanum;lycopersicum",
                        "Plantae;Solanaceae;Solanum;tuberosum",
                        "Plantae;Solanaceae;Solanum;unknown",
                        "Plantae;Solanaceae;Solanum;uncultured",
                        "Plantae;UNIDENTIFIED"))
filter_ambiguous_taxa(obj)
```

`filter_obs` *Filter observations with a list of conditions*

Description

Filter data in a [taxmap()] object (in 'obj$data') with a set of conditions. See [dplyr::filter()] for the inspiration for this function and more information. Calling the function using the 'obj$filter_obs(...)' style edits "obj" in place, unlike most R functions. However, calling the function using the 'filter_obs(obj, ...)' imitates R's traditional copy-on-modify semantics, so "obj" would not be changed; instead a changed version would be returned, like most R functions.

```
obj$filter_obs(data, ..., drop_taxa = FALSE, drop_obs = TRUE,
               subtaxa = FALSE, supertaxa = TRUE, reassign_obs = FALSE)
filter_obs(obj, data, ..., drop_taxa = FALSE, drop_obs = TRUE,
           subtaxa = FALSE, supertaxa = TRUE, reassign_obs = FALSE)
```

Arguments

`obj`	An object of type [taxmap()]
`data`	Dataset names, indexes, or a logical vector that indicates which datasets in 'obj$data' to filter. If multiple datasets are filterd at once, then they must be the same length.
`...`	One or more filtering conditions. Any variable name that appears in [all_names()] can be used as if it was a vector on its own. Each filtering condition can be one of two things: * 'integer': One or more dataset indexes. * 'logical': A 'TRUE'/'FALSE' vector of length equal to the number of items in the dataset.
`drop_taxa`	('logical' of length 1) If 'FALSE', preserve taxa even if all of their observations are filtered out. If 'TRUE', remove taxa for which all observations were filtered out. Note that only taxa that are unobserved due to this filtering will be removed; there might be other taxa without observations to begin with that will not be removed.
`drop_obs`	('logical') This only has an effect when 'drop_taxa' is 'TRUE'. When 'TRUE', observations for other data sets (i.e. not 'data') assigned to taxa that are removed when filtering 'data' are also removed. Otherwise, only data for taxa that are not present in all other data sets will be removed. This option can be either simply 'TRUE'/'FALSE', meaning that all data sets will be treated the same, or a logical vector can be supplied with names corresponding one or more data

sets in ‘obj$data‘. For example, ‘c(abundance = TRUE, stats = FALSE)‘ would remove observations in ‘obj$data$abundance‘, but not in ‘obj$data$stats‘.

`subtaxa` (‘logical‘ or ‘numeric‘ of length 1) This only has an effect when ‘drop_taxa‘ is ‘TRUE‘. If ‘TRUE‘, include subtaxa of taxa passing the filter. Positive numbers indicate the number of ranks below the target taxa to return. ‘0‘ is equivalent to ‘FALSE‘. Negative numbers are equivalent to ‘TRUE‘.

`supertaxa` (‘logical‘ or ‘numeric‘ of length 1) This only has an effect when ‘drop_taxa‘ is ‘TRUE‘. If ‘TRUE‘, include supertaxa of taxa passing the filter. Positive numbers indicate the number of ranks above the target taxa to return. ‘0‘ is equivalent to ‘FALSE‘. Negative numbers are equivalent to ‘TRUE‘.

`reassign_obs` (‘logical‘) This only has an effect when ‘drop_taxa‘ is ‘TRUE‘. If ‘TRUE‘, observations assigned to removed taxa will be reassigned to the closest supertaxon that passed the filter. If there are no supertaxa of such an observation that passed the filter, they will be filtered out if ‘drop_obs‘ is ‘TRUE‘. This option can be either simply ‘TRUE‘/‘FALSE‘, meaning that all data sets will be treated the same, or a logical vector can be supplied with names corresponding one or more data sets in ‘obj$data‘. For example, ‘c(abundance = TRUE, stats = FALSE)‘ would reassign observations in ‘obj$data$abundance‘, but not in ‘obj$data$stats‘.

`target` DEPRECIATED. use "data" instead.

Value

An object of type [taxmap()]

See Also

Other taxmap manipulation functions: `arrange_obs()`, `arrange_taxa()`, `filter_taxa()`, `mutate_obs()`, `sample_frac_obs()`, `sample_frac_taxa()`, `sample_n_obs()`, `sample_n_taxa()`, `select_obs()`, `transmute_obs()`

Examples

```
# Filter by row index
filter_obs(ex_taxmap, "info", 1:2)

# Filter by TRUE/FALSE
filter_obs(ex_taxmap, "info", dangerous == FALSE)
filter_obs(ex_taxmap, "info", dangerous == FALSE, n_legs > 0)
filter_obs(ex_taxmap, "info", n_legs == 2)

# Remove taxa whose obserservations were filtered out
filter_obs(ex_taxmap, "info", n_legs == 2, drop_taxa = TRUE)

# Preserve other data sets while removing taxa
filter_obs(ex_taxmap, "info", n_legs == 2, drop_taxa = TRUE,
           drop_obs = c(abund = FALSE))

# When filtering taxa, do not return supertaxa of taxa that are preserved
```

```
filter_obs(ex_taxmap, "info", n_legs == 2, drop_taxa = TRUE,
           supertaxa = FALSE)

# Filter multiple datasets at once
filter_obs(ex_taxmap, c("info", "phylopic_ids", "foods"), n_legs == 2)
```

`filter_taxa` *Filter taxa with a list of conditions*

Description

Filter taxa in a [taxonomy()] or [taxmap()] object with a series of conditions. Any variable name that appears in [all_names()] can be used as if it was a vector on its own. See [dplyr::filter()] for the inspiration for this function and more information. Calling the function using the ‘obj$filter_taxa(...)‘ style edits "obj" in place, unlike most R functions. However, calling the function using the ‘filter_taxa(obj, ...)‘ imitates R’s traditional copy-on-modify semantics, so "obj" would not be changed; instead a changed version would be returned, like most R functions.

```
filter_taxa(obj, ..., subtaxa = FALSE, supertaxa = FALSE,
  drop_obs = TRUE, reassign_obs = TRUE, reassign_taxa = TRUE,
  invert = FALSE, keep_order = TRUE)
obj$filter_taxa(..., subtaxa = FALSE, supertaxa = FALSE,
  drop_obs = TRUE, reassign_obs = TRUE, reassign_taxa = TRUE,
  invert = FALSE, keep_order = TRUE)
```

Arguments

`obj`	An object of class [taxonomy()] or [taxmap()]
`...`	One or more filtering conditions. Any variable name that appears in [all_names()] can be used as if it was a vector on its own. Each filtering condition must resolve to one of three things: * ‘character‘: One or more taxon IDs contained in ‘obj$edge_list$to‘ * ‘integer‘: One or more row indexes of ‘obj$edge_list‘ * ‘logical‘: A ‘TRUE‘/‘FALSE‘ vector of length equal to the number of rows in ‘obj$edge_list‘ * ‘NULL‘: ignored
`subtaxa`	(‘logical‘ or ‘numeric‘ of length 1) If ‘TRUE‘, include subtaxa of taxa passing the filter. Positive numbers indicate the number of ranks below the target taxa to return. ‘0‘ is equivalent to ‘FALSE‘. Negative numbers are equivalent to ‘TRUE‘.
`supertaxa`	(‘logical‘ or ‘numeric‘ of length 1) If ‘TRUE‘, include supertaxa of taxa passing the filter. Positive numbers indicate the number of ranks above the target taxa to return. ‘0‘ is equivalent to ‘FALSE‘. Negative numbers are equivalent to ‘TRUE‘.
`drop_obs`	(‘logical‘) This option only applies to [taxmap()] objects. If ‘FALSE‘, include observations (i.e. user-defined data in ‘obj$data‘) even if the taxon they are assigned to is filtered out. Observations assigned to removed taxa will be assigned

to NA. This option can be either simply 'TRUE'/'FALSE', meaning that all data sets will be treated the same, or a logical vector can be supplied with names corresponding one or more data sets in 'obj$data'. For example, 'c(abundance = FALSE, stats = TRUE)' would include observations whose taxon was filtered out in 'obj$data$abundance', but not in 'obj$data$stats'. See the 'reassign_obs' option below for further complications.

`reassign_obs` ('logical' of length 1) This option only applies to [taxmap()] objects. If 'TRUE', observations (i.e. user-defined data in 'obj$data') assigned to removed taxa will be reassigned to the closest supertaxon that passed the filter. If there are no supertaxa of such an observation that passed the filter, they will be filtered out if 'drop_obs' is 'TRUE'. This option can be either simply 'TRUE'/'FALSE', meaning that all data sets will be treated the same, or a logical vector can be supplied with names corresponding one or more data sets in 'obj$data'. For example, 'c(abundance = TRUE, stats = FALSE)' would reassign observations in 'obj$data$abundance', but not in 'obj$data$stats'.

`reassign_taxa` ('logical' of length 1) If 'TRUE', subtaxa of removed taxa will be reassigned to the closest supertaxon that passed the filter. This is useful for removing intermediate levels of a taxonomy.

`invert` ('logical' of length 1) If 'TRUE', do NOT include the selection. This is different than just replacing a '==' with a '!=' because this option negates the selection after taking into account the 'subtaxa' and 'supertaxa' options. This is useful for removing a taxon and all its subtaxa for example.

`keep_order` ('logical' of length 1) If 'TRUE', keep relative order of taxa not filtered out. For example, the result of 'filter_taxa(ex_taxmap, 1:3)' and 'filter_taxa(ex_taxmap, 3:1)' would be the same. Does not affect dataset order, only taxon order. This is useful for maintaining order correspondence with a dataset that has one value per taxon.

Value

An object of type [taxonomy()] or [taxmap()]

See Also

Other taxmap manipulation functions: `arrange_obs()`, `arrange_taxa()`, `filter_obs()`, `mutate_obs()`, `sample_frac_obs()`, `sample_frac_taxa()`, `sample_n_obs()`, `sample_n_taxa()`, `select_obs()`, `transmute_obs()`

Examples

```
# Filter by index
filter_taxa(ex_taxmap, 1:3)

# Filter by taxon ID
filter_taxa(ex_taxmap, c("b", "c", "d"))

# Fiter by TRUE/FALSE
filter_taxa(ex_taxmap, taxon_names == "Plantae", subtaxa = TRUE)
filter_taxa(ex_taxmap, n_obs > 3)
```

```
filter_taxa(ex_taxmap, ! taxon_ranks %in% c("species", "genus"))
filter_taxa(ex_taxmap, taxon_ranks == "genus", n_obs > 1)

# Filter by an observation characteristic
dangerous_taxa <- sapply(ex_taxmap$obs("info"),
                         function(i) any(ex_taxmap$data$info$dangerous[i]))
filter_taxa(ex_taxmap, dangerous_taxa)

# Include supertaxa
filter_taxa(ex_taxmap, 12, supertaxa = TRUE)
filter_taxa(ex_taxmap, 12, supertaxa = 2)

# Include subtaxa
filter_taxa(ex_taxmap, 1, subtaxa = TRUE)
filter_taxa(ex_taxmap, 1, subtaxa = 2)

# Dont remove rows in user-defined data corresponding to removed taxa
filter_taxa(ex_taxmap, 2, drop_obs = FALSE)
filter_taxa(ex_taxmap, 2, drop_obs = c(info = FALSE))

# Remove a taxon and it subtaxa
filter_taxa(ex_taxmap, taxon_names == "Mammalia",
            subtaxa = TRUE, invert = TRUE)
```

`get_data` *Get data in a taxmap object by name*

Description

Given a vector of names, return a list of data (usually lists/vectors) contained in a [taxonomy()] or [taxmap()] object. Each item will be named by taxon ids when possible.

```
obj$get_data(name = NULL, ...)
get_data(obj, name = NULL, ...)
```

Arguments

`obj`	A [taxonomy()] or [taxmap()] object
`name`	(‘character‘) Names of data to return. If not supplied, return all data listed in [all_names()].
`...`	Passed to [all_names()]. Used to filter what kind of data is returned (e.g. columns in tables or function output?) if ‘name‘ is not supplied or what kinds are allowed if ‘name‘ is supplied.

Value

‘list‘ of vectors or lists. Each vector or list will be named by associated taxon ids if possible.

See Also

Other NSE helpers: `all_names()`, `data_used`, `names_used`

Examples

```
# Get specific values
get_data(ex_taxmap, c("reaction", "n_legs", "taxon_ranks"))

# Get all values
get_data(ex_taxmap)
```

`get_dataset` *Get a data set from a taxmap object*

Description

Get a data set from a taxmap object and complain if it does not exist.

Arguments

`obj`	A taxmap object
`data`	Dataset name, index, or a logical vector that indicates which dataset in 'obj$data' to add columns to.

Examples

```
# Get data set by name
get_dataset(ex_taxmap, "info")

# Get data set by indeex_taxmap
get_dataset(ex_taxmap, 1)

# Get data set by T/F vector
get_dataset(ex_taxmap, startsWith(names(ex_taxmap$data), "i"))
```

`get_data_frame` *Get data in a taxonomy or taxmap object by name*

Description

Given a vector of names, return a table of the indicated data contained in a [taxonomy()] or [taxmap()] object.

```
obj$get_data_frame(name = NULL, ...)
get_data_frame(obj, name = NULL, ...)
```

Arguments

obj	A [taxonomy()] or [taxmap()] object
name	(‘character‘) Names of data to return. If not supplied, return all data listed in [all_names()].
...	Passed to [all_names()]. Used to filter what kind of data is returned (e.g. columns in tables or function output?) if ‘name‘ is not supplied or what kinds are allowed if ‘name‘ is supplied.

Details

Note: This function will not work with variables in datasets in [taxmap()] objects unless their rows correspond 1:1 with all taxa.

Value

‘data.frame‘

Examples

```
# Get specific values
get_data_frame(ex_taxmap, c("taxon_names", "taxon_indexes", "is_stem"))
```

---

heat_tree *Plot a taxonomic tree*

Description

Plots the distribution of values associated with a taxonomic classification/heirarchy. Taxonomic classifications can have multiple roots, resulting in multiple trees on the same plot. A tree consists of elements, element properties, conditions, and mapping properties which are represented as parameters in the heat_tree object. The elements (e.g. nodes, edges, lables, and individual trees) are the infrastructure of the heat tree. The element properties (e.g. size and color) are characteristics that are manipulated by various data conditions and mapping properties. The element properties can be explicitly defined or automatically generated. The conditions are data (e.g. taxon statistics, such as abundance) represented in the taxmap/metacoder object. The mapping properties are parameters (e.g. transformations, range, interval, and layout) used to change the elements/element properties and how they are used to represent (or not represent) the various conditions.

Usage

```
heat_tree(...)

## S3 method for class 'Taxmap'
heat_tree(.input, ...)
```

```
## Default S3 method:
heat_tree(
  taxon_id,
  supertaxon_id,
  node_label = NA,
  edge_label = NA,
  tree_label = NA,
  node_size = 1,
  edge_size = node_size,
  node_label_size = node_size,
  edge_label_size = edge_size,
  tree_label_size = as.numeric(NA),
  node_color = "#999999",
  edge_color = node_color,
  tree_color = NA,
  node_label_color = "#000000",
  edge_label_color = "#000000",
  tree_label_color = "#000000",
  node_size_trans = "area",
  edge_size_trans = node_size_trans,
  node_label_size_trans = node_size_trans,
  edge_label_size_trans = edge_size_trans,
  tree_label_size_trans = "area",
  node_color_trans = "area",
  edge_color_trans = node_color_trans,
  tree_color_trans = "area",
  node_label_color_trans = "area",
  edge_label_color_trans = "area",
  tree_label_color_trans = "area",
  node_size_range = c(NA, NA),
  edge_size_range = c(NA, NA),
  node_label_size_range = c(NA, NA),
  edge_label_size_range = c(NA, NA),
  tree_label_size_range = c(NA, NA),
  node_color_range = quantative_palette(),
  edge_color_range = node_color_range,
  tree_color_range = quantative_palette(),
  node_label_color_range = quantative_palette(),
  edge_label_color_range = quantative_palette(),
  tree_label_color_range = quantative_palette(),
  node_size_interval = range(node_size, na.rm = TRUE, finite = TRUE),
  node_color_interval = NULL,
  edge_size_interval = range(edge_size, na.rm = TRUE, finite = TRUE),
  edge_color_interval = NULL,
  node_label_max = 500,
  edge_label_max = 500,
  tree_label_max = 500,
  overlap_avoidance = 1,
```

```
  margin_size = c(0, 0, 0, 0),
  layout = "reingold-tilford",
  initial_layout = "fruchterman-reingold",
  make_node_legend = TRUE,
  make_edge_legend = TRUE,
  title = NULL,
  title_size = 0.08,
  node_legend_title = "Nodes",
  edge_legend_title = "Edges",
  node_color_axis_label = NULL,
  node_size_axis_label = NULL,
  edge_color_axis_label = NULL,
  edge_size_axis_label = NULL,
  node_color_digits = 3,
  node_size_digits = 3,
  edge_color_digits = 3,
  edge_size_digits = 3,
  background_color = "#FFFFFF00",
  output_file = NULL,
  aspect_ratio = 1,
  repel_labels = TRUE,
  repel_force = 1,
  repel_iter = 1000,
  verbose = FALSE,
  ...
)
```

Arguments

`...`	(other named arguments) Passed to the `igraph` layout function used.
`.input`	An object of type `taxmap`
`taxon_id`	The unique ids of taxa.
`supertaxon_id`	The unique id of supertaxon `taxon_id` is a part of.
`node_label`	See details on labels. Default: no labels.
`edge_label`	See details on labels. Default: no labels.
`tree_label`	See details on labels. The label to display above each graph. The value of the root of each graph will be used. Default: None.
`node_size`	See details on size. Default: constant size.
`edge_size`	See details on size. Default: relative to node size.
`node_label_size`	See details on size. Default: relative to vertex size.
`edge_label_size`	See details on size. Default: relative to edge size.
`tree_label_size`	See details on size. Default: relative to graph size.
`node_color`	See details on colors. Default: grey.

`edge_color` See details on colors. Default: same as node color.

`tree_color` See details on colors. The value of the root of each graph will be used. Overwrites the node and edge color if specified. Default: Not used.

`node_label_color`
See details on colors. Default: black.

`edge_label_color`
See details on colors. Default: black.

`tree_label_color`
See details on colors. Default: black.

`node_size_trans`
See details on transformations. Default: `"area"`.

`edge_size_trans`
See details on transformations. Default: same as `node_size_trans`.

`node_label_size_trans`
See details on transformations. Default: same as `node_size_trans`.

`edge_label_size_trans`
See details on transformations. Default: same as `edge_size_trans`.

`tree_label_size_trans`
See details on transformations. Default: `"area"`.

`node_color_trans`
See details on transformations. Default: `"area"`.

`edge_color_trans`
See details on transformations. Default: same as node color transformation.

`tree_color_trans`
See details on transformations. Default: `"area"`.

`node_label_color_trans`
See details on transformations. Default: `"area"`.

`edge_label_color_trans`
See details on transformations. Default: `"area"`.

`tree_label_color_trans`
See details on transformations. Default: `"area"`.

`node_size_range`
See details on ranges. Default: Optimize to balance overlaps and range size.

`edge_size_range`
See details on ranges. Default: relative to node size range.

`node_label_size_range`
See details on ranges. Default: relative to node size.

`edge_label_size_range`
See details on ranges. Default: relative to edge size.

`tree_label_size_range`
See details on ranges. Default: relative to tree size.

`node_color_range`
See details on ranges. Default: Color-blind friendly palette.

`edge_color_range`
See details on ranges. Default: same as node color.

`tree_color_range`
: See details on ranges. Default: Color-blind friendly palette.

`node_label_color_range`
: See details on ranges. Default: Color-blind friendly palette.

`edge_label_color_range`
: See details on ranges. Default: Color-blind friendly palette.

`tree_label_color_range`
: See details on ranges. Default: Color-blind friendly palette.

`node_size_interval`
: See details on intervals. Default: The range of values in `node_size`.

`node_color_interval`
: See details on intervals. Default: The range of values in `node_color`.

`edge_size_interval`
: See details on intervals. Default: The range of values in `edge_size`.

`edge_color_interval`
: See details on intervals. Default: The range of values in `edge_color`.

`node_label_max` The maximum number of node labels. Default: 20.

`edge_label_max` The maximum number of edge labels. Default: 20.

`tree_label_max` The maximum number of tree labels. Default: 20.

`overlap_avoidance`
: (`numeric`) The relative importance of avoiding overlaps vs maximizing size range. Higher numbers will cause node size optimization to avoid overlaps more. Default: `1`.

`margin_size` (`numeric` of length 2) The horizontal and vertical margins. c(left, right, bottom, top). Default: `0, 0, 0, 0`.

`layout` The layout algorithm used to position nodes. See details on layouts. Default: `"reingold-tilford"`.

`initial_layout` he layout algorithm used to set the initial position of nodes, passed as input to the `layout` algorithm. See details on layouts. Default: Not used.

`make_node_legend`
: if TRUE, make legend for node size/color mappings.

`make_edge_legend`
: if TRUE, make legend for edge size/color mappings.

`title` Name to print above the graph.

`title_size` The size of the title relative to the rest of the graph.

`node_legend_title`
: The title of the legend for node data. Can be 'NA' or 'NULL' to remove the title.

`edge_legend_title`
: The title of the legend for edge data. Can be 'NA' or 'NULL' to remove the title.

`node_color_axis_label`
: The label on the scale axis corresponding to `node_color`. Default: The expression given to `node_color`.

`node_size_axis_label`
: The label on the scale axis corresponding to `node_size`. Default: The expression given to `node_size`.

`edge_color_axis_label`
: The label on the scale axis corresponding to `edge_color`. Default: The expression given to `edge_color`.

`edge_size_axis_label`
: The label on the scale axis corresponding to `edge_size`. Default: The expression given to `edge_size`.

`node_color_digits`
: The number of significant figures used for the numbers on the scale axis corresponding to `node_color`. Default: 3.

`node_size_digits`
: The number of significant figures used for the numbers on the scale axis corresponding to `node_size`. Default: 3.

`edge_color_digits`
: The number of significant figures used for the numbers on the scale axis corresponding to `edge_color`. Default: 3.

`edge_size_digits`
: The number of significant figures used for the numbers on the scale axis corresponding to `edge_size`. Default: 3.

`background_color`
: The background color of the plot. Default: Transparent

`output_file`
: The path to one or more files to save the plot in using `ggplot2::ggsave`. The type of the file will be determined by the extension given. Default: Do not save plot.

`aspect_ratio`
: The aspect_ratio of the plot.

`repel_labels`
: If `TRUE` (Default), use the ggrepel package to spread out labels.

`repel_force`
: The force of which overlapping labels will be repelled from eachother.

`repel_iter`
: The number of iterations used when repelling labels

`verbose`
: If `TRUE` print progress reports as the function runs.

labels

The labels of nodes, edges, and trees can be added. Node labels are centered over their node. Edge labels are displayed over edges, in the same orientation. Tree labels are displayed over their tree.

Accepts a vector, the same length `taxon_id` or a factor of its length.

sizes

The size of nodes, edges, labels, and trees can be mapped to various conditions. This is useful for displaying statistics for taxa, such as abundance. Only the relative size of the condition is used, not the values themselves. The <element>_size_trans (transformation) parameter can be used to make the size mapping non-linear. The <element>_size_range parameter can be used to proportionately change the size of an element based on the condition mapped to that element. The

<element>_size_interval parameter can be used to change the limit at which a condition will be graphically represented as the same size as the minimum/maximum <element>_size_range.

Accepts a `numeric` vector, the same length `taxon_id` or a factor of its length.

colors

The colors of nodes, edges, labels, and trees can be mapped to various conditions. This is useful for visually highlighting/clustering groups of taxa. Only the relative size of the condition is used, not the values themselves. The <element>_color_trans (transformation) parameter can be used to make the color mapping non-linear. The <element>_color_range parameter can be used to proportionately change the color of an element based on the condition mapped to that element. The <element>_color_interval parameter can be used to change the limit at which a condition will be graphically represented as the same color as the minimum/maximum <element>_color_range.

Accepts a vector, the same length `taxon_id` or a factor of its length. If a numeric vector is given, it is mapped to a color scale. Hex values or color names can be used (e.g. `#000000` or `"black"`).

Mapping Properties

transformations

Before any conditions specified are mapped to an element property (color/size), they can be transformed to make the mapping non-linear. Any of the transformations listed below can be used by specifying their name. A customized function can also be supplied to do the transformation.

"linear" Proportional to radius/diameter of node

"area" circular area; better perceptual accuracy than `"linear"`

"log10" Log base 10 of radius

"log2" Log base 2 of radius

"ln" Log base e of radius

"log10 area" Log base 10 of circular area

"log2 area" Log base 2 of circular area

"ln area" Log base e of circular area

ranges

The displayed range of colors and sizes can be explicitly defined or automatically generated. When explicitly used, the size range will proportionately increase/decrease the size of a particular element. Size ranges are specified by supplying a `numeric` vector with two values: the minimum and maximum. The units used should be between 0 and 1, representing the proportion of a dimension of the graph. Since the dimensions of the graph are determined by layout, and not always square, the value that 1 corresponds to is the square root of the graph area (i.e. the side of a square with the same area as the plotted space). Color ranges can be any number of color values as either HEX codes (e.g. `#000000`) or color names (e.g. `"black"`).

layout

Layouts determine the position of node elements on the graph. They are implemented using the `igraph` package. Any additional arguments passed to `heat_tree` are passed to the `igraph` function used. The following `character` values are understood:

"automatic" Use `igraph::nicely`. Let `igraph` choose the layout.

"reingold-tilford" Use `igraph::as_tree`. A circular tree-like layout.

"davidson-harel" Use `igraph::with_dh`. A type of simulated annealing.

"gem" Use `igraph::with_gem`. A force-directed layout.

"graphopt" Use `igraph::with_graphopt`. A force-directed layout.

"mds" Use `igraph::with_mds`. Multidimensional scaling.

"fruchterman-reingold" Use `igraph::with_fr`. A force-directed layout.

"kamada-kawai" Use `igraph::with_kk`. A layout based on a physical model of springs.

"large-graph" Use `igraph::with_lgl`. Meant for larger graphs.

"drl" Use `igraph::with_drl`. A force-directed layout.

intervals

This is the minimum and maximum of values displayed on the legend scales. Intervals are specified by supplying a `numeric` vector with two values: the minimum and maximum. When explicitly used, the <element>_<property>_interval will redefine the way the actual conditional values are being represented by setting a limit for the <element>_<property>. Any condition below the minimum <element>_<property>_interval will be graphically represented the same as a condition AT the minimum value in the full range of conditional values. Any value above the maximum <element>_<property>_interval will be graphically represented the same as a value AT the maximum value in the full range of conditional values. By default, the minimum and maximum equals the <element>_<property>_range used to infer the value of the <element>_<property>. Setting a custom interval is useful for making <element>_<properties> in multiple graphs correspond to the same conditions, or setting logical boundaries (such as `c(0,1)` for proportions. Note that this is different from the <element>_<property>_range mapping property, which determines the size/color of graphed elements.

Acknowledgements

This package includes code from the R package ggrepel to handle label overlap avoidance with permission from the author of ggrepel Kamil Slowikowski. We included the code instead of depending on ggrepel because we are using internal functions to ggrepel that might change in the future. We thank Kamil Slowikowski for letting us use his code and would like to acknowledge his implementation of the label overlap avoidance used in metacoder.

Examples

```
# Parse dataset for plotting
x = parse_tax_data(hmp_otus, class_cols = "lineage", class_sep = ";",
                   class_key = c(tax_rank = "taxon_rank", tax_name = "taxon_name"),
                   class_regex = "^(.+)__(.+)$")
```

```
# Default appearance:
#  No parmeters are needed, but the default tree is not too useful
heat_tree(x)

# A good place to start:
#  There will always be "taxon_names" and "n_obs" variables, so this is a
#  good place to start. This will shown the number of OTUs in this case.
heat_tree(x, node_label = taxon_names, node_size = n_obs, node_color = n_obs)

# Plotting read depth:
#  To plot read depth, you first need to add up the number of reads per taxon.
#  The function `calc_taxon_abund` is good for this.
x$data$taxon_counts <- calc_taxon_abund(x, data = "tax_data")
x$data$taxon_counts$total <- rowSums(x$data$taxon_counts[, -1]) # -1 = taxon_id column
heat_tree(x, node_label = taxon_names, node_size = total, node_color = total)

# Plotting multiple variables:
#  You can plot up to 4 quantative variables use node/edge size/color, but it
#  is usually best to use 2 or 3. The plot below uses node size for number of
#  OTUs and color for number of reads and edge size for number of samples
x$data$n_samples <- calc_n_samples(x, data = "taxon_counts")
heat_tree(x, node_label = taxon_names, node_size = n_obs, node_color = total,
          edge_color = n_samples)

# Different layouts:
#  You can use any layout implemented by igraph. You can also specify an
#  initial layout to seed the main layout with.
heat_tree(x, node_label = taxon_names, node_size = n_obs, node_color = n_obs,
          layout = "davidson-harel")
heat_tree(x, node_label = taxon_names, node_size = n_obs, node_color = n_obs,
          layout = "davidson-harel", initial_layout = "reingold-tilford")

# Axis labels:
#  You can add custom labeles to the legends
heat_tree(x, node_label = taxon_names, node_size = n_obs, node_color = total,
          edge_color = n_samples, node_size_axis_label = "Number of OTUs",
          node_color_axis_label = "Number of reads",
          edge_color_axis_label = "Number of samples")

# Overlap avoidance:
#  You can change how much node overlap avoidance is used.
heat_tree(x, node_label = taxon_names, node_size = n_obs, node_color = n_obs,
          overlap_avoidance = .5)

# Label overlap avoidance
#  You can modfiy how label scattering is handled using the `replel_force` and
# `repel_iter` options. You can turn off label scattering using the `repel_labels` option.
heat_tree(x, node_label = taxon_names, node_size = n_obs, node_color = n_obs,
          repel_force = 2, repel_iter = 20000)
heat_tree(x, node_label = taxon_names, node_size = n_obs, node_color = n_obs,
          repel_labels = FALSE)
```

```
# Setting the size of graph elements:
#  You can force nodes, edges, and lables to be a specific size/color range instead
#  of letting the function optimize it. These options end in `_range`.
heat_tree(x, node_label = taxon_names, node_size = n_obs, node_color = n_obs,
          node_size_range = c(0.01, .1))
heat_tree(x, node_label = taxon_names, node_size = n_obs, node_color = n_obs,
          edge_color_range = c("black", "#FFFFFF"))
heat_tree(x, node_label = taxon_names, node_size = n_obs, node_color = n_obs,
          node_label_size_range = c(0.02, 0.02))

# Setting the transformation used:
#  You can change how raw statistics are converted to color/size using options
#  ending in _trans.
heat_tree(x, node_label = taxon_names, node_size = n_obs, node_color = n_obs,
          node_size_trans = "log10 area")

# Setting the interval displayed:
#  By default, the whole range of the statistic provided will be displayed.
#  You can set what range of values are displayed using options ending in `_interval`.
heat_tree(x, node_label = taxon_names, node_size = n_obs, node_color = n_obs,
          node_size_interval = c(10, 100))
```

---

heat_tree_matrix *Plot a matrix of heat trees*

Description

Plot a matrix of heat trees for showing pairwise comparisons. A larger, labelled tree serves as a key for the matrix of smaller unlabelled trees. The data for this function is typically created with compare_groups,

Usage

```
heat_tree_matrix(
  obj,
  data,
  label_small_trees = FALSE,
  key_size = 0.6,
  seed = 1,
  output_file = NULL,
  row_label_color = diverging_palette()[3],
  col_label_color = diverging_palette()[1],
  row_label_size = 12,
  col_label_size = 12,
  ...,
  dataset = NULL
)
```

Arguments

obj	A taxmap object
data	The name of a table in obj$data that is the output of compare_groups or in the same format.
label_small_trees	If TRUE add labels to small trees as well as the key tree. Otherwise, only the key tree will be labeled.
key_size	The size of the key tree relative to the whole graph. For example, 0.5 means half the width/height of the graph.
seed	That random seed used to make the graphs.
output_file	The path to one or more files to save the plot in using ggsave. The type of the file will be determined by the extension given. Default: Do not save plot.
row_label_color	The color of the row labels on the right side of the matrix. Default: based on the node_color_range.
col_label_color	The color of the columns labels along the top of the matrix. Default: based on the node_color_range.
row_label_size	The size of the row labels on the right side of the matrix. Default: 12.
col_label_size	The size of the columns labels along the top of the matrix. Default: 12.
...	Passed to heat_tree. Some options will be overwritten.
dataset	DEPRECIATED. use "data" instead.

Examples

```
# Parse dataset for plotting
x <- parse_tax_data(hmp_otus, class_cols = "lineage", class_sep = ";",
                    class_key = c(tax_rank = "taxon_rank", tax_name = "taxon_name"),
                    class_regex = "^(.+)__(.+)$")

# Convert counts to proportions
x$data$otu_table <- calc_obs_props(x, data = "tax_data", cols = hmp_samples$sample_id)

# Get per-taxon counts
x$data$tax_table <- calc_taxon_abund(x, data = "otu_table", cols = hmp_samples$sample_id)

# Calculate difference between treatments
x$data$diff_table <- compare_groups(x, data = "tax_table",
                                    cols = hmp_samples$sample_id,
                                    groups = hmp_samples$body_site)

# Plot results (might take a few minutes)
heat_tree_matrix(x,
                 data = "diff_table",
                 node_size = n_obs,
                 node_label = taxon_names,
                 node_color = log2_median_ratio,
```

```
                node_color_range = diverging_palette(),
                node_color_trans = "linear",
                node_color_interval = c(-3, 3),
                edge_color_interval = c(-3, 3),
                node_size_axis_label = "Number of OTUs",
                node_color_axis_label = "Log2 ratio median proportions")
```

`hierarchies` *Make a set of many [hierarchy()] class objects*

Description

NOTE: This will soon be depreciated. Make a set of many [hierarchy()] class objects. This is just a thin wrapper over a standard list.

Usage

```
hierarchies(..., .list = NULL)
```

Arguments

`...`	Any number of object of class [hierarchy()]
`.list`	Any number of object of class [hierarchy()] in a list

Value

An 'R6Class' object of class [hierarchy()]

See Also

Other classes: `hierarchy()`, `taxa()`, `taxmap()`, `taxon()`, `taxon_database()`, `taxon_id()`, `taxon_name()`, `taxon_rank()`, `taxonomy()`

`hierarchy` *The Hierarchy class*

Description

A class containing an ordered list of [taxon()] objects that represent a hierarchical classification.

Usage

```
hierarchy(..., .list = NULL)
```

Arguments

...	Any number of object of class 'Taxon' or taxonomic names as character strings
.list	An alternate to the '...' input. Any number of object of class [taxon()] or character vectors in a list. Cannot be used with '...'.

Details

On initialization, taxa are sorted if they have ranks with a known order.

Methods

'pop(rank_names)' Remove 'Taxon' elements by rank name, taxon name or taxon ID. The change happens in place, so you don't need to assign output to a new object. returns self - rank_names (character) a vector of rank names

'pick(rank_names)' Select 'Taxon' elements by rank name, taxon name or taxon ID. The change happens in place, so you don't need to assign output to a new object. returns self - rank_names (character) a vector of rank names

Value

An 'R6Class' object of class 'Hierarchy'

See Also

Other classes: hierarchies(), taxa(), taxmap(), taxon(), taxon_database(), taxon_id(), taxon_name(), taxon_rank(), taxonomy()

Examples

```
(x <- taxon(
  name = taxon_name("Poaceae"),
  rank = taxon_rank("family"),
  id = taxon_id(4479)
))

(y <- taxon(
  name = taxon_name("Poa"),
  rank = taxon_rank("genus"),
  id = taxon_id(4544)
))

(z <- taxon(
  name = taxon_name("Poa annua"),
  rank = taxon_rank("species"),
  id = taxon_id(93036)
))

(res <- hierarchy(z, y, x))

res$taxa
res$ranklist
```

```
# null taxa
x <- taxon(NULL)
(res <- hierarchy(x, x, x))
## similar to hierarchy(), but `taxa` slot is not empty
```

`highlight_taxon_ids` *Highlight taxon ID column*

Description

Changes the font of a taxon ID column in a table print out.

Usage

```
highlight_taxon_ids(table_text, header_index, row_indexes)
```

Arguments

`table_text`	The print out of the table in a character vector, one element per line.
`header_index`	The row index that contains the table column names
`row_indexes`	The indexes of the rows to be formatted.

`hmp_otus` *A HMP subset*

Description

A subset of the Human Microbiome Project abundance matrix produced by QIIME. It contains OTU ids, taxonomic lineages, and the read counts for 50 samples. See `hmp_samples` for the matching dataset of sample information.

Format

A 1,000 x 52 tibble.

Details

The 50 samples were randomly selected such that there were 10 in each of 5 treatments: "Saliva", "Throat", "Stool", "Right_Antecubital_fossa", "Anterior_nares". For each treatment, there were 5 samples from men and 5 from women.

Source

Subset from data available at https://www.hmpdacc.org/hmp/HMQCP/

See Also

Other hmp_data: `hmp_samples`

hmp_samples *Sample information for HMP subset*

Description

The sample information for a subset of the Human Microbiome Project data. It contains the sample ID, sex, and body site for each sample in the abundance matrix stored in hmp_otus. The "sample_id" column corresponds to the column names of hmp_otus.

Format

A 50 x 3 tibble.

Details

The 50 samples were randomly selected such that there were 10 in each of 5 treatments: "Saliva", "Throat", "Stool", "Right_Antecubital_fossa", "Anterior_nares". For each treatment, there were 5 samples from men and 5 from women. "Right_Antecubital_fossa" was renamed to "Skin" and "Anterior_nares" to "Nose".

Source

Subset from data available at https://www.hmpdacc.org/hmp/HMQCP/

See Also

Other hmp_data: hmp_otus

id_classifications *Get ID classifications of taxa*

Description

Get classification strings of taxa in an object of type [taxonomy()] or [taxmap()] composed of taxon IDs. Each classification is constructed by concatenating the taxon ids of the given taxon and its supertaxa.

```
obj$id_classifications(sep = ";")
id_classifications(obj, sep = ";")
```

Arguments

obj	([taxonomy()] or [taxmap()])
sep	(‘character‘ of length 1) The character(s) to place between taxon IDs

Value

'character'

See Also

Other taxonomy data functions: `classifications()`, `is_branch()`, `is_internode()`, `is_leaf()`, `is_root()`, `is_stem()`, `map_data()`, `map_data_()`, `n_leaves()`, `n_leaves_1()`, `n_subtaxa()`, `n_subtaxa_1()`, `n_supertaxa()`, `n_supertaxa_1()`, `taxon_ids()`, `taxon_indexes()`, `taxon_names()`, `taxon_ranks()`

Examples

```
# Get classifications of IDs for each taxon
id_classifications(ex_taxmap)

# Use a different seperator
id_classifications(ex_taxmap, sep = '|')
```

---

internodes *Get "internode" taxa*

Description

Return the "internode" taxa for a [taxonomy()] or [taxmap()] object. An internode is any taxon with a single immediate supertaxon and a single immediate subtaxon. They can be removed from a tree without any loss of information on the relative relationship between remaining taxa. Can also be used to get the internodes of a subset of taxa.

```
obj$internodes(subset = NULL, value = "taxon_indexes")
internodes(obj, subset = NULL, value = "taxon_indexes")
```

Arguments

`obj`	The [taxonomy()] or [taxmap()] object containing taxon information to be queried.
`subset`	Taxon IDs, TRUE/FALSE vector, or taxon indexes used to subset the tree prior to determining internodes. Default: All taxa in 'obj' will be used. Any variable name that appears in [all_names()] can be used as if it was a vector on its own. Note that internodes are determined after the filtering, so a given taxon might be a internode on the unfiltered tree, but not a internode on the filtered tree.
`value`	What data to return. This is usually the name of column in a table in 'obj$data'. Any result of [all_names()] can be used, but it usually only makes sense to use data that corresponds to taxa 1:1, such as [taxon_ranks()]. By default, taxon indexes are returned.

Value

'character'

See Also

Other taxonomy indexing functions: branches(), leaves(), roots(), stems(), subtaxa(), supertaxa()

Examples

```
# Return indexes of branch taxa
internodes(ex_taxmap)

# Return indexes for a subset of taxa
internodes(ex_taxmap, subset = 2:17)
internodes(ex_taxmap, subset = n_obs > 1)

# Return something besides taxon indexes
internodes(ex_taxmap, value = "taxon_names")
```

---

is_ambiguous *Find ambiguous taxon names*

Description

Find taxa with ambiguous names, such as "unknown" or "uncultured".

Usage

```
is_ambiguous(
  taxon_names,
  unknown = TRUE,
  uncultured = TRUE,
  name_regex = ".",
  ignore_case = TRUE
)
```

Arguments

taxon_names	A taxmap object
unknown	If TRUE, Remove taxa with names the suggest they are placeholders for unknown taxa (e.g. "unknown ...").
uncultured	If TRUE, Remove taxa with names the suggest they are assigned to uncultured organisms (e.g. "uncultured ...").
name_regex	The regex code to match a valid character in a taxon name. For example, "[a-z]" would mean taxon names can only be lower case letters.
ignore_case	If TRUE, dont consider the case of the text when determining a match.

Details

If you encounter a taxon name that represents an ambiguous taxon that is not filtered out by this function, let us know and we will add it.

Value

TRUE/FALSE vector corresponding to `taxon_names`

Examples

```
is_ambiguous(c("unknown", "uncultured", "homo sapiens", "kfdsjfdljsdf"))
```

`is_branch` *Test if taxa are branches*

Description

Test if taxa are branches in a [taxonomy()] or [taxmap()] object. Branches are taxa in the interior of the tree that are not [roots()], [stems()], or [leaves()].

```
obj$is_branch()
is_branch(obj)
```

Arguments

`obj`	The [taxonomy()] or [taxmap()] object.

Value

A 'logical' of length equal to the number of taxa.

See Also

Other taxonomy data functions: `classifications()`, `id_classifications()`, `is_internode()`, `is_leaf()`, `is_root()`, `is_stem()`, `map_data()`, `map_data_()`, `n_leaves()`, `n_leaves_1()`, `n_subtaxa()`, `n_subtaxa_1()`, `n_supertaxa()`, `n_supertaxa_1()`, `taxon_ids()`, `taxon_indexes()`, `taxon_names()`, `taxon_ranks()`

Examples

```
# Test which taxon IDs correspond to branches
is_branch(ex_taxmap)

# Filter out branches
filter_taxa(ex_taxmap, ! is_branch)
```

is_internode *Test if taxa are "internodes"*

Description

Test if taxa are "internodes" in a [taxonomy()] or [taxmap()] object. An internode is any taxon with a single immediate supertaxon and a single immediate subtaxon. They can be removed from a tree without any loss of information on the relative relationship between remaining taxa.

```
obj$is_internode()
is_internode(obj)
```

Arguments

obj	The [taxonomy()] or [taxmap()] object.

Value

A 'logical' of length equal to the number of taxa.

See Also

Other taxonomy data functions: classifications(), id_classifications(), is_branch(), is_leaf(), is_root(), is_stem(), map_data(), map_data_(), n_leaves(), n_leaves_1(), n_subtaxa(), n_subtaxa_1(), n_supertaxa(), n_supertaxa_1(), taxon_ids(), taxon_indexes(), taxon_names(), taxon_ranks()

Examples

```
# Test for which taxon IDs correspond to internodes
is_internode(ex_taxmap)

# Filter out internodes
filter_taxa(ex_taxmap, ! is_internode)
```

is_leaf *Test if taxa are leaves*

Description

Test if taxa are leaves in a [taxonomy()] or [taxmap()] object. Leaves are taxa without subtaxa, typically species.

```
obj$is_leaf()
is_leaf(obj)
```

Arguments

obj	The [taxonomy()] or [taxmap()] object.

Value

A ‘logical‘ of length equal to the number of taxa.

See Also

Other taxonomy data functions: classifications(), id_classifications(), is_branch(), is_internode(), is_root(), is_stem(), map_data(), map_data_(), n_leaves(), n_leaves_1(), n_subtaxa(), n_subtaxa_1(), n_supertaxa(), n_supertaxa_1(), taxon_ids(), taxon_indexes(), taxon_names(), taxon_ranks()

Examples

```
# Test which taxon IDs correspond to leaves
is_leaf(ex_taxmap)

# Filter out leaves
filter_taxa(ex_taxmap, ! is_leaf)
```

---

is_root *Test if taxa are roots*

Description

Test if taxa are roots in a [taxonomy()] or [taxmap()] object. Roots are taxa without supertaxa, typically things like "Bacteria", or "Life".

```
obj$is_root()
is_root(obj)
```

Arguments

obj	The [taxonomy()] or [taxmap()] object.

Value

A ‘logical‘ of length equal to the number of taxa.

See Also

Other taxonomy data functions: classifications(), id_classifications(), is_branch(), is_internode(), is_leaf(), is_stem(), map_data(), map_data_(), n_leaves(), n_leaves_1(), n_subtaxa(), n_subtaxa_1(), n_supertaxa(), n_supertaxa_1(), taxon_ids(), taxon_indexes(), taxon_names(), taxon_ranks()

Examples

```
# Test for which taxon IDs correspond to roots
is_root(ex_taxmap)

# Filter out roots
filter_taxa(ex_taxmap, ! is_root)
```

`is_stem` *Test if taxa are stems*

Description

Test if taxa are stems in a [taxonomy()] or [taxmap()] object. Stems are taxa from the [roots()] taxa to the first taxon with more than one subtaxon. These can usually be filtered out of the taxonomy without removing any information on how the remaining taxa are related.

```
obj$is_stem()
is_stem(obj)
```

Arguments

`obj`	The [taxonomy()] or [taxmap()] object.

Value

A 'logical' of length equal to the number of taxa.

See Also

Other taxonomy data functions: `classifications()`, `id_classifications()`, `is_branch()`, `is_internode()`, `is_leaf()`, `is_root()`, `map_data()`, `map_data_()`, `n_leaves()`, `n_leaves_1()`, `n_subtaxa()`, `n_subtaxa_1()`, `n_supertaxa()`, `n_supertaxa_1()`, `taxon_ids()`, `taxon_indexes()`, `taxon_names()`, `taxon_ranks()`

Examples

```
# Test which taxon IDs correspond to stems
is_stem(ex_taxmap)

# Filter out stems
filter_taxa(ex_taxmap, ! is_stem)
```

layout_functions *Layout functions*

Description

Functions used to determine graph layout. Calling the function with no parameters returns available function names. Calling the function with only the name of a function returns that function. Supplying a name and a graph object to run the layout function on the graph.

Usage

```
layout_functions(
  name = NULL,
  graph = NULL,
  intitial_coords = NULL,
  effort = 1,
  ...
)
```

Arguments

name	(character of length 1 OR NULL) name of algorithm. Leave NULL to see all options.
graph	(igraph) The graph to generate the layout for.
intitial_coords	(matrix) Initial node layout to base new layout off of.
effort	(numeric of length 1) The amount of effort to put into layouts. Typically determines the the number of iterations.
...	(other arguments) Passed to igraph layout function used.

Value

The name available functions, a layout functions, or a two-column matrix depending on how arguments are provided.

Examples

```
# List available function names:
layout_functions()

# Execute layout function on graph:
layout_functions("davidson-harel", igraph::make_ring(5))
```

`leaves` *Get leaf taxa*

Description

Return the leaf taxa for a [taxonomy()] or [taxmap()] object. Leaf taxa are taxa with no subtaxa.

```
obj$leaves(subset = NULL, recursive = TRUE, simplify = FALSE, value = "taxon_indexes")
leaves(obj, subset = NULL, recursive = TRUE, simplify = FALSE, value = "taxon_indexes")
```

Arguments

`obj`	The [taxonomy()] or [taxmap()] object containing taxon information to be queried.
`subset`	Taxon IDs, TRUE/FALSE vector, or taxon indexes to find leaves for. Default: All taxa in 'obj' will be used. Any variable name that appears in [all_names()] can be used as if it was a vector on its own.
`recursive`	('logical' or 'numeric') If 'FALSE', only return the leaves if they occur one rank below the target taxa. If 'TRUE', return all of the leaves for each taxon. Positive numbers indicate the number of recursions (i.e. number of ranks below the target taxon to return). '1' is equivalent to 'FALSE'. Negative numbers are equivalent to 'TRUE'.
`simplify`	('logical') If 'TRUE', then combine all the results into a single vector of unique values.
`value`	What data to return. This is usually the name of column in a table in 'obj$data'. Any result of 'all_names(obj)' can be used, but it usually only makes sense to data that corresponds to taxa 1:1, such as [taxon_ranks()]. By default, taxon indexes are returned.

Value

'character'

See Also

Other taxonomy indexing functions: `branches()`, `internodes()`, `roots()`, `stems()`, `subtaxa()`, `supertaxa()`

Examples

```
# Return indexes of leaf taxa
leaves(ex_taxmap)

# Return indexes for a subset of taxa
leaves(ex_taxmap, subset = 2:17)
leaves(ex_taxmap, subset = taxon_names == "Plantae")

# Return something besides taxon indexes
```

```
leaves(ex_taxmap, value = "taxon_names")
leaves(ex_taxmap, subset = taxon_ranks == "genus", value = "taxon_names")

# Return a vector of all unique values
leaves(ex_taxmap, value = "taxon_names", simplify = TRUE)

# Only return leaves for their direct supertaxa
leaves(ex_taxmap, value = "taxon_names", recursive = FALSE)
```

`leaves_apply` *Apply function to leaves of each taxon*

Description

Apply a function to the leaves of each taxon. This is similar to using [leaves()] with [lapply()] or [sapply()].

```
obj$leaves_apply(func, subset = NULL, recursive = TRUE,
  simplify = FALSE, value = "taxon_indexes", ...)
leaves_apply(obj, func, subset = NULL, recursive = TRUE,
  simplify = FALSE, value = "taxon_indexes", ...)
```

Arguments

`obj`	The [taxonomy()] or [taxmap()] object containing taxon information to be queried.
`func`	(‘function‘) The function to apply.
`subset`	Taxon IDs, TRUE/FALSE vector, or taxon indexes to use. Default: All taxa in ‘obj‘ will be used. Any variable name that appears in [all_names()] can be used as if it was a vector on its own.
`recursive`	(‘logical‘ or ‘numeric‘) If ‘FALSE‘, only return the leaves if they occur one rank below the target taxa. If ‘TRUE‘, return all of the leaves for each taxon. Positive numbers indicate the number of recursions (i.e. number of ranks below the target taxon to return). ‘1‘ is equivalent to ‘FALSE‘. Negative numbers are equivalent to ‘TRUE‘.
`simplify`	(‘logical‘) If ‘TRUE‘, then combine all the results into a single vector of unique values.
`value`	What data to give to the function. Any result of ‘all_names(obj)‘ can be used, but it usually only makes sense to use data that has an associated taxon id.
`...`	Extra arguments are passed to the function ‘func‘.

Examples

```
# Count number of leaves under each taxon or its subtaxa
leaves_apply(ex_taxmap, length)

# Count number of leaves under each taxon
leaves_apply(ex_taxmap, length, recursive = FALSE)

# Converting output of leaves to upper case
leaves_apply(ex_taxmap, value = "taxon_names", toupper)

# Passing arguments to the function
leaves_apply(ex_taxmap, value = "taxon_names", paste0, collapse = ", ")
```

---

lookup_tax_data *Convert one or more data sets to taxmap*

Description

Looks up taxonomic data from NCBI sequence IDs, taxon IDs, or taxon names that are present in a table, list, or vector. Also can incorporate additional associated datasets.

Usage

```
lookup_tax_data(
  tax_data,
  type,
  column = 1,
  datasets = list(),
  mappings = c(),
  database = "ncbi",
  include_tax_data = TRUE,
  use_database_ids = TRUE,
  ask = TRUE
)
```

Arguments

tax_data	A table, list, or vector that contain sequence IDs, taxon IDs, or taxon names. * tables: The ‘column‘ option must be used to specify which column contains the sequence IDs, taxon IDs, or taxon names. * lists: There must be only one item per list entry unless the ‘column‘ option is used to specify what item to use in each list entry. * vectors: simply a vector of sequence IDs, taxon IDs, or taxon names.
type	What type of information can be used to look up the classifications. Takes one of the following values: * ‘"seq_id"‘: A database sequence ID with an associated classification (e.g. NCBI accession numbers). * ‘"taxon_id"‘: A reference

database taxon ID (e.g. a NCBI taxon ID) * ‘"taxon_name"‘: A single taxon name (e.g. "Homo sapiens" or "Primates") * ‘"fuzzy_name"‘: A single taxon name, but check for misspellings first. Only use if you think there are misspellings. Using ‘"taxon_name"‘ is faster.

`column` (‘character‘ or ‘integer‘) The name or index of the column that contains information used to lookup classifications. This only applies when a table or list is supplied to ‘tax_data‘.

`datasets` Additional lists/vectors/tables that should be included in the resulting ‘taxmap‘ object. The ‘mappings‘ option is use to specify how these data sets relate to the ‘tax_data‘ and, by inference, what taxa apply to each item.

`mappings` (named ‘character‘) This defines how the taxonomic information in ‘tax_data‘ applies to data in ‘datasets‘. This option should have the same number of inputs as ‘datasets‘, with values corresponding to each dataset. The names of the character vector specify what information in ‘tax_data‘ is shared with info in each ‘dataset‘, which is specified by the corresponding values of the character vector. If there are no shared variables, you can add ‘NA‘ as a placeholder, but you could just leave that data out since it is not benefiting from being in the taxmap object. The names/values can be one of the following: * For tables, the names of columns can be used. * ‘"{{index}}"‘ : This means to use the index of rows/items * ‘"{{name}}"‘ : This means to use row/item names. * ‘"{{value}}"‘ : This means to use the values in vectors or lists. Lists will be converted to vectors using [unlist()].

`database` (‘character‘) The name of a database to use to look up classifications. Options include "ncbi", "itis", "eol", "col", "tropicos", and "nbn".

`include_tax_data`
(‘TRUE‘/‘FALSE‘) Whether or not to include ‘tax_data‘ as a dataset, like those in ‘datasets‘.

`use_database_ids`
(‘TRUE‘/‘FALSE‘) Whether or not to use downloaded database taxon ids instead of arbitrary, automatically-generated taxon ids.

`ask` (‘TRUE‘/‘FALSE‘) Whether or not to prompt the user for input. Currently, this would only happen when looking up the taxonomy of a taxon name with multiple matches. If ‘FALSE‘, taxa with multiple hits are treated as if they do not exist in the database. This might change in the future if we can find an elegant way of handling this.

Failed Downloads

If you have invalid inputs or a download fails for another reason, then there will be a "unknown" taxon ID as a placeholder and failed inputs will be assigned to this ID. You can remove these using [filter_taxa()] like so: ‘filter_taxa(result, taxon_ids != "unknown")‘. Add ‘drop_obs = FALSE‘ if you want the input data, but want to remove the taxon.

See Also

Other parsers: `extract_tax_data()`, `parse_dada2()`, `parse_edge_list()`, `parse_greengenes()`, `parse_mothur_tax_summary()`, `parse_mothur_taxonomy()`, `parse_newick()`, `parse_phylo()`,

parse_phyloseq(), parse_qiime_biom(), parse_rdp(), parse_silva_fasta(), parse_tax_data(), parse_ubiome(), parse_unite_general()

Examples

```
# Look up taxon names in vector from NCBI
lookup_tax_data(c("homo sapiens", "felis catus", "Solanaceae"),
                type = "taxon_name")

# Look up taxon names in list from NCBI
lookup_tax_data(list("homo sapiens", "felis catus", "Solanaceae"),
                type = "taxon_name")

# Look up taxon names in table from NCBI
my_table <- data.frame(name = c("homo sapiens", "felis catus"),
                       decency = c("meh", "good"))
lookup_tax_data(my_table, type = "taxon_name", column = "name")

# Look up taxon names from a different database
lookup_tax_data(c("homo sapiens", "felis catus", "Solanaceae"),
                type = "taxon_name", database = "ITIS")

# Prevent asking questions for ambiguous taxon names
lookup_tax_data(c("homo sapiens", "felis catus", "Solanaceae"),
                type = "taxon_name", database = "ITIS", ask = FALSE)

# Look up taxon IDs from NCBI
lookup_tax_data(c("9689", "9694", "9643"), type = "taxon_id")

# Look up sequence IDs from NCBI
lookup_tax_data(c("AB548412", "FJ358423", "DQ334818"),
                type = "seq_id")

# Make up new taxon IDs instead of using the downloaded ones
lookup_tax_data(c("AB548412", "FJ358423", "DQ334818"),
                type = "seq_id", use_database_ids = FALSE)


# --- Parsing multiple datasets at once (advanced) ---
# The rest is one example for how to classify multiple datasets at once.

# Make example data with taxonomic classifications
species_data <- data.frame(tax = c("Mammalia;Carnivora;Felidae",
                                   "Mammalia;Carnivora;Felidae",
                                   "Mammalia;Carnivora;Ursidae"),
                           species = c("Panthera leo",
                                       "Panthera tigris",
                                       "Ursus americanus"),
                           species_id = c("A", "B", "C"))

# Make example data associated with the taxonomic data
# Note how this does not contain classifications, but
# does have a varaible in common with "species_data" ("id" = "species_id")
```

```
abundance <- data.frame(id = c("A", "B", "C", "A", "B", "C"),
                        sample_id = c(1, 1, 1, 2, 2, 2),
                        counts = c(23, 4, 3, 34, 5, 13))

# Make another related data set named by species id
common_names <- c(A = "Lion", B = "Tiger", C = "Bear", "Oh my!")

# Make another related data set with no names
foods <- list(c("ungulates", "boar"),
              c("ungulates", "boar"),
              c("salmon", "fruit", "nuts"))

# Make a taxmap object with these three datasets
x = lookup_tax_data(species_data,
                    type = "taxon_name",
                    datasets = list(counts = abundance,
                                    my_names = common_names,
                                    foods = foods),
                    mappings = c("species_id" = "id",
                                 "species_id" = "{{name}}",
                                 "{{index}}" = "{{index}}"),
                    column = "species")

# Note how all the datasets have taxon ids now
x$data

# This allows for complex mappings between variables that other functions use
map_data(x, my_names, foods)
map_data(x, counts, my_names)
```

`make_dada2_asv_table` *Make a imitation of the dada2 ASV abundance matrix*

Description

Attempts to save the abundance matrix stored as a table in a taxmap object in the dada2 ASV abundance matrix format. If the taxmap object was created using `parse_dada2`, then it should be able to replicate the format exactly with the default settings.

Usage

```
make_dada2_asv_table(obj, asv_table = "asv_table", asv_id = "asv_id")
```

Arguments

`obj`	A taxmap object
`asv_table`	The name of the abundance matrix in the taxmap object to use.
`asv_id`	The name of the column in `asv_table` with unique ASV ids or sequences.

Value

A numeric `matrix` with rows as samples and columns as ASVs

See Also

Other writers: `make_dada2_tax_table()`, `write_greengenes()`, `write_mothur_taxonomy()`, `write_rdp()`, `write_silva_fasta()`, `write_unite_general()`

`make_dada2_tax_table` *Make a imitation of the dada2 taxonomy matrix*

Description

Attempts to save the taxonomy information assocaited with an abundance matrix in a taxmap object in the dada2 taxonomy matrix format. If the taxmap object was created using `parse_dada2`, then it should be able to replicate the format exactly with the default settings.

Usage

```
make_dada2_tax_table(obj, asv_table = "asv_table", asv_id = "asv_id")
```

Arguments

`obj`	A taxmap object
`asv_table`	The name of the abundance matrix in the taxmap object to use.
`asv_id`	The name of the column in `asv_table` with unique ASV ids or sequences.

Value

A character `matrix` with rows as ASVs and columns as taxonomic ranks.

See Also

Other writers: `make_dada2_asv_table()`, `write_greengenes()`, `write_mothur_taxonomy()`, `write_rdp()`, `write_silva_fasta()`, `write_unite_general()`

`map_data` *Create a mapping between two variables*

Description

Creates a named vector that maps the values of two variables associated with taxa in a [taxonomy()] or [taxmap()] object. Both values must be named by taxon ids.

```
obj$map_data(from, to, warn = TRUE)
map_data(obj, from, to, warn = TRUE)
```

Arguments

`obj`	The [taxonomy()] or [taxmap()] object.
`from`	The value used to name the output. There will be one output value for each value in 'from'. Any variable that appears in [all_names()] can be used as if it was a variable on its own.
`to`	The value returned in the output. Any variable that appears in [all_names()] can be used as if it was a variable on its own.
`warn`	If 'TRUE', issue a warning if there are multiple unique values of 'to' for each value of 'from'.

Value

A vector of 'to' values named by values in 'from'.

See Also

Other taxonomy data functions: `classifications()`, `id_classifications()`, `is_branch()`, `is_internode()`, `is_leaf()`, `is_root()`, `is_stem()`, `map_data_()`, `n_leaves()`, `n_leaves_1()`, `n_subtaxa()`, `n_subtaxa_1()`, `n_supertaxa()`, `n_supertaxa_1()`, `taxon_ids()`, `taxon_indexes()`, `taxon_names()`, `taxon_ranks()`

Examples

```
# Mapping between two variables in `all_names(ex_taxmap)`
map_data(ex_taxmap, from = taxon_names, to = n_legs > 0)

# Mapping with external variables
x = c("d" = "looks like a cat", "h" = "big scary cats",
      "i" = "smaller cats", "m" = "might eat you", "n" = "Meow! (Feed me!)")
map_data(ex_taxmap, from = taxon_names, to = x)
```

`map_data_` *Create a mapping without NSE*

Description

Creates a named vector that maps the values of two variables associated with taxa in a [taxonomy()] or [taxmap()] object without using Non-Standard Evaluation (NSE). Both values must be named by taxon ids. This is the same as [map_data()] without NSE and can be useful in some odd cases where NSE fails to work as expected.

```
obj$map_data(from, to)
map_data(obj, from, to)
```

Arguments

`obj`	The [taxonomy()] or [taxmap()] object.
`from`	The value used to name the output. There will be one output value for each value in 'from'.
`to`	The value returned in the output.

Value

A vector of 'to' values named by values in 'from'.

See Also

Other taxonomy data functions: `classifications()`, `id_classifications()`, `is_branch()`, `is_internode()`, `is_leaf()`, `is_root()`, `is_stem()`, `map_data()`, `n_leaves()`, `n_leaves_1()`, `n_subtaxa()`, `n_subtaxa_1()`, `n_supertaxa()`, `n_supertaxa_1()`, `taxon_ids()`, `taxon_indexes()`, `taxon_names()`, `taxon_ranks()`

Examples

```
x = c("d" = "looks like a cat", "h" = "big scary cats",
      "i" = "smaller cats", "m" = "might eat you", "n" = "Meow! (Feed me!)")
map_data_(ex_taxmap, from = ex_taxmap$taxon_names(), to = x)
```

`metacoder` *Metacoder*

Description

A package for planning and analysis of amplicon metagenomics research projects.

Details

The goal of the `metacoder` package is to provide a set of tools for:

- Standardized parsing of taxonomic information from diverse resources.
- Visualization of statistics distributed over taxonomic classifications.
- Evaluating potential metabarcoding primers for taxonomic specificity.
- Providing flexible functions for analyzing taxonomic and abundance data.

To accomplish these goals, `metacoder` leverages resources from other R packages, interfaces with external programs, and provides novel functions where needed to allow for entire analyses within R.

Documentation

The full documentation can be found online at https://grunwaldlab.github.io/metacoder_documentation/.

There is also a short vignette included for offline use that can be accessed by the following code:

```
browseVignettes(package = "metacoder")
```

Plotting:

- `heat_tree`
- `heat_tree_matrix`

In silico PCR:

- `primersearch`

Analysis:

- `calc_taxon_abund`
- `calc_obs_props`
- `rarefy_obs`
- `compare_groups`
- `zero_low_counts`
- `calc_n_samples`
- `filter_ambiguous_taxa`

Parsers:

- parse_greengenes
- parse_mothur_tax_summary
- parse_mothur_taxonomy
- parse_newick
- parse_phyloseq
- parse_phylo
- parse_qiime_biom
- parse_rdp
- parse_silva_fasta
- parse_unite_general

Writers:

- write_greengenes
- write_mothur_taxonomy
- write_rdp
- write_silva_fasta
- write_unite_general

Database querying:

- ncbi_taxon_sample

Main classes

These are the classes users would typically interact with:

* [taxon]: A class used to define a single taxon. Many other classes in the 'taxa" package include one or more objects of this class. * : Stores one or more [taxon] objects. This is just a thin wrapper for a list of [taxon] objects. * [hierarchy]: A class containing an ordered list of [taxon] objects that represent a hierarchical classification. * [hierarchies]: A list of taxonomic classifications. This is just a thin wrapper for a list of [hierarchy] objects. * [taxonomy]: A taxonomy composed of [taxon] objects organized in a tree structure. This differs from the [hierarchies] class in how the [taxon] objects are stored. Unlike a [hierarchies] object, each unique taxon is stored only once and the relationships between taxa are stored in an edgelist. * [taxmap]: A class designed to store a taxonomy and associated user-defined data. This class builds on the [taxonomy] class. User defined data can be stored in the list 'obj$data', where 'obj' is a taxmap object. Any number of user-defined lists, vectors, or tables mapped to taxa can be manipulated in a cohesive way such that relationships between taxa and data are preserved.

Minor classes

These classes are mostly components for the larger classes above and would not typically be used on their own.

* [taxon_database]: Used to store information about taxonomy databases. * [taxon_id]: Used to store taxon IDs, either arbitrary or from a particular taxonomy database. * [taxon_name]: Used to store taxon names, either arbitrary or from a particular taxonomy database. * [taxon_rank]: Used to store taxon ranks (e.g. species, family), either arbitrary or from a particular taxonomy database.

Major manipulation functions

These are some of the more important functions used to filter data in classes that store multiple taxa, like [hierarchies], [taxmap], and [taxonomy].

* [filter_taxa]: Filter taxa in a [taxonomy] or [taxmap] object with a series of conditions. Relationships between remaining taxa and user-defined data are preserved (There are many options controlling this). * [filter_obs]: Filter user-defined data [taxmap] object with a series of conditions. Relationships between remaining taxa and user-defined data are preserved (There are many options controlling this); * [sample_n_taxa]: Randomly sample taxa. Has same abilities as [filter_taxa]. * [sample_n_obs]: Randomly sample observations. Has same abilities as [filter_obs]. * [mutate_obs]: Add datasets or columns to datasets in the ‘data‘ list of [taxmap] objects. * [pick]: Pick out specific taxa, while others are dropped in [hierarchy] and [hierarchies] objects. * [pop]: Pop out taxa (drop them) in [hierarchy] and [hierarchies] objects. * [span]: Select a range of taxa, either by two names, or relational operators in [hierarchy] and [hierarchies] objects.

Mapping functions

There are lots of functions for getting information for each taxon.

* [subtaxa]: Return data for the subtaxa of each taxon in an [taxonomy] or [taxmap] object. * [supertaxa]: Return data for the supertaxa of each taxon in an [taxonomy] or [taxmap] object. * [roots]: Return data for the roots of each taxon in an [taxonomy] or [taxmap] object. * [leaves]: Return data for the leaves of each taxon in an [taxonomy] or [taxmap] object. * [obs]: Return user-specific data for each taxon and all of its subtaxa in an [taxonomy] or [taxmap] object.

The kind of classes used

Note, this is mostly of interest to developers and advanced users.

The classes in the ‘taxa‘ package are mostly [R6](https://adv-r.hadley.nz/r6.html) classes ([R6Class]). A few of the simpler ones (and [hierarchies]) are [S3](https://adv-r.hadley.nz/s3.html) instead. R6 classes are different than most R objects because they are [mutable](https://en.wikipedia.org/wiki/Immutable_object) (e.g. A function can change its input without returning it). In this, they are more similar to class systems in [object-oriented](https://en.wikipedia.org/wiki/Object-oriented_programming) languages like python. As in other object-oriented class systems, functions are thought to "belong" to classes (i.e. the data), rather than functions existing independently of the data. For example, the function ‘print‘ in R exists apart from what it is printing, although it will change how it prints based on what the class of the data is that is passed to it. In fact, a user can make a custom print method for their own class by defining a function called ‘print.myclassname‘. In contrast, the functions that operate on R6 functions are "packaged" with the data they operate on. For example, a print method of an object for an R6 class might be called like ‘my_data$print()‘ instead of ‘print(my_data)‘.

The two ways to call functions

Note, you will need to read the previous section to fully understand this one.

Since the R6 function syntax (e.g. ‘my_data$print()‘) might be confusing to many R users, all functions in ‘taxa‘ also have S3 versions. For example, the [filter_taxa()] function can be called on a [taxmap] object called ‘my_obj‘ like ‘my_obj$filter_taxa(...)‘ (the R6 syntax) or ‘filter_taxa(my_obj, ...)‘ (the S3 syntax). For some functions, these two way of calling the function can have different effect. For functions that do not returned a modified version of the input (e.g. [subtaxa()]), the two ways have identical behavior. However, functions like [filter_taxa()], that modify their inputs, actually change the object passed to them as the first argument as well as returning that object. For example,

‘my_obj <- filter_taxa(my_obj, ...)‘

and

‘my_obj$filter_taxa(...)‘

and

‘new_obj <- my_obj$filter_taxa(...)‘

all replace ‘my_obj‘ with the filtered result, but

‘new_obj <- filter_taxa(my_obj, ...)‘

will not modify ‘my_obj‘.

Non-standard evaluation

This is a rather advanced topic.

Like packages such as ‘ggplot2‘ and [dplyr], the ‘taxa‘ package uses non-standard evaluation to allow code to be more readable and shorter. In effect, there are variables that only "exist" inside a function call and depend on what is passed to that function as the first parameter (usually a class object). For example, in the ‘dpylr‘ function [filter()], column names can be used as if they were independent variables. See ‘?dpylr::filter‘ for examples of this. The ‘taxa‘ package builds on this idea.

For many functions that work on [taxonomy] or [taxmap] objects (e.g. [filter_taxa]), some functions that return per-taxon information (e.g. [taxon_names()]) can be referred to by just the name of the function. When one of these functions are referred to by name, the function is run on the relevant object and its value replaces the function name. For example,

‘new_obj <- filter_taxa(my_obj, taxon_names == "Bacteria")‘

is identical to:

‘new_obj <- filter_taxa(my_obj, taxon_names(my_obj) == "Bacteria")‘

which is identical to:

‘new_obj <- filter_taxa(my_obj, my_obj$taxon_names() == "Bacteria")‘

which is identical to:

‘my_names <- taxon_names(my_obj)‘

‘new_obj <- filter_taxa(my_obj, my_names == "Bacteria")‘

For ‘taxmap‘ objects, you can also use names of user defined lists, vectors, and the names of columns in user-defined tables that are stored in the ‘obj$data‘ list. See [filter_taxa()] for examples.

You can even add your own functions that are called by name by adding them to the ‘obj$funcs‘ list. For any object with functions that use non-standard evaluation, you can see what values can be used with [all_names()] like ‘all_names(obj)‘.

Dependencies and inspiration

Various elements of the ‘taxa‘ package were inspired by the [dplyr] and [taxize] packages. This package started as parts of the ‘metacoder‘ and ‘binomen‘ packages. There are also many dependencies that make ‘taxa‘ possible.

Feedback and contributions

Find a problem? Have a suggestion? Have a question? Please submit an issue at our [GitHub repository](https://github.com/ropensci/taxa):

https://github.com/ropensci/taxa/issues

A GitHub account is free and easy to set up. We welcome feedback! If you don’t want to use GitHub for some reason, feel free to email us. We do prefer posting to github since it allows others that might have the same issue to see our conversation. It also helps us keep track of what problems we need to address.

Want to contribute code or make a change to the code? Great, thank you! Please [fork](https://help.github.com/articles/fork-a-repo/) our GitHub repository and submit a [pull request](https://help.github.com/articles/about-pull-requests/).

Author(s)

Zachary Foster and Niklaus Grunwald

`mutate_obs` *Add columns to [taxmap()] objects*

Description

Add columns to tables in ‘obj$data‘ in [taxmap()] objects. See [dplyr::mutate()] for the inspiration for this function and more information. Calling the function using the ‘obj$mutate_obs(...)‘ style edits "obj" in place, unlike most R functions. However, calling the function using the ‘mutate_obs(obj, ...)‘ imitates R’s traditional copy-on-modify semantics, so "obj" would not be changed; instead a changed version would be returned, like most R functions.

```
obj$mutate_obs(data, ...)
mutate_obs(obj, data, ...)
```

Arguments

obj	An object of type [taxmap()]
data	Dataset name, index, or a logical vector that indicates which dataset in 'obj$data' to add columns to.
...	One or more named columns to add. Newly created columns can be referenced in the same function call. Any variable name that appears in [all_names()] can be used as if it was a vector on its own.
target	DEPRECIATED. use "data" instead.

Value

An object of type [taxmap()]

See Also

Other taxmap manipulation functions: `arrange_obs()`, `arrange_taxa()`, `filter_obs()`, `filter_taxa()`, `sample_frac_obs()`, `sample_frac_taxa()`, `sample_n_obs()`, `sample_n_taxa()`, `select_obs()`, `transmute_obs()`

Examples

```
# Add column to existing tables
mutate_obs(ex_taxmap, "info",
           new_col = "Im new",
           newer_col = paste0(new_col, "er!"))

# Create columns in a new table
mutate_obs(ex_taxmap, "new_table",
           nums = 1:10,
           squared = nums ^ 2)

# Add a new vector
mutate_obs(ex_taxmap, "new_vector", 1:10)

# Add a new list
mutate_obs(ex_taxmap, "new_list", list(1, 2))
```

---

ncbi_taxon_sample *Download representative sequences for a taxon*

Description

Downloads a sample of sequences meant to evenly capture the diversity of a given taxon. Can be used to get a shallow sampling of vast groups. **CAUTION:** This function can make MANY queries to Genbank depending on arguments given and can take a very long time. Choose your arguments carefully to avoid long waits and needlessly stressing NCBI's servers. Use a downloaded database and a parser from the `taxa` package when possible.

Usage

```
ncbi_taxon_sample(
  name = NULL,
  id = NULL,
  target_rank,
  min_counts = NULL,
  max_counts = NULL,
  interpolate_min = TRUE,
  interpolate_max = TRUE,
  min_children = NULL,
  max_children = NULL,
  seqrange = "1:3000",
  getrelated = FALSE,
  fuzzy = TRUE,
  limit = 10,
  entrez_query = NULL,
  hypothetical = FALSE,
  verbose = TRUE
)
```

Arguments

name	(character of length 1) The taxon to download a sample of sequences for.
id	(character of length 1) The taxon id to download a sample of sequences for.
target_rank	(character of length 1) The finest taxonomic rank at which to sample. The finest rank at which replication occurs. Must be a finer rank than taxon.
min_counts	(named numeric) The minimum number of sequences to download for each taxonomic rank. The names correspond to taxonomic ranks.
max_counts	(named numeric) The maximum number of sequences to download for each taxonomic rank. The names correspond to taxonomic ranks.
interpolate_min	(logical) If TRUE, values supplied to min_counts and min_children will be used to infer the values of intermediate ranks not specified. Linear interpolation between values of specified ranks will be used to determine values of unspecified ranks.
interpolate_max	(logical) If TRUE, values supplied to max_counts and max_children will be used to infer the values of intermediate ranks not specified. Linear interpolation between values of specified ranks will be used to determine values of unspecified ranks.
min_children	(named numeric) The minimum number sub-taxa of taxa for a given rank must have for its sequences to be searched. The names correspond to taxonomic ranks.
max_children	(named numeric) The maximum number sub-taxa of taxa for a given rank must have for its sequences to be searched. The names correspond to taxonomic ranks.

seqrange	(character) Sequence range, as e.g., "1:1000". This is the range of sequence lengths to search for. So "1:1000" means search for sequences from 1 to 1000 characters in length.
getrelated	(logical) If TRUE, gets the longest sequences of a species in the same genus as the one searched for. If FALSE, returns nothing if no match found.
fuzzy	(logical) Whether to do fuzzy taxonomic ID search or exact search. If TRUE, we use xXarbitraryXx[porgn:__txid<ID>], but if FALSE, we use txid<ID>. Default: FALSE
limit	(numeric) Number of sequences to search for and return. Max of 10,000. If you search for 6000 records, and only 5000 are found, you will of course only get 5000 back.
entrez_query	(character; length 1) An Entrez-format query to filter results with. This is useful to search for sequences with specific characteristics. The format is the same as the one used to seach genbank. (https://www.ncbi.nlm.nih.gov/books/NBK3837/#EntrezHelp.Entrez_Searching_Options)
hypothetical	(logical; length 1) If FALSE, an attempt will be made to not return hypothetical or predicted sequences judging from accession number prefixs (XM and XR). This can result in less than the limit being returned even if there are more sequences available, since this filtering is done after searching NCBI.
verbose	(logical) If TRUE, progress messages will be printed.

Examples

```
if (requireNamespace("traits", quietly = TRUE)) {
  # Look up 5 ITS sequences from each fungal class
  data <- ncbi_taxon_sample(name = "Fungi", target_rank = "class", limit = 5,
                            entrez_query = '"internal transcribed spacer"[All Fields]')

  # Look up taxonomic information for sequences
  obj <- lookup_tax_data(data, type = "seq_id", column = "gi_no")

  # Plot information
  metacoder::filter_taxa(obj, taxon_names == "Fungi", subtaxa = TRUE) %>%
    heat_tree(node_label = taxon_names, node_color = n_obs, node_size = n_obs)
}
```

---

n_leaves *Get number of leaves*

Description

Get number of leaves for each taxon in an object of type [taxonomy()] or [taxmap()]

```
obj$n_leaves()
n_leaves(obj)
```

Arguments

obj	([taxonomy()] or [taxmap()])

Value

numeric

See Also

Other taxonomy data functions: classifications(), id_classifications(), is_branch(), is_internode(), is_leaf(), is_root(), is_stem(), map_data(), map_data_(), n_leaves_1(), n_subtaxa(), n_subtaxa_1(), n_supertaxa(), n_supertaxa_1(), taxon_ids(), taxon_indexes(), taxon_names(), taxon_ranks()

Examples

```
# Get number of leaves for each taxon
n_leaves(ex_taxmap)

# Filter taxa based on number of leaves
filter_taxa(ex_taxmap, n_leaves > 0)
```

---

n_leaves_1 *Get number of leaves*

Description

Get number of leaves for each taxon in an object of type [taxonomy()] or [taxmap()], not including leaves of subtaxa etc.

```
obj$n_leaves_1()
n_leaves_1(obj)
```

Arguments

obj	([taxonomy()] or [taxmap()])

Value

numeric

See Also

Other taxonomy data functions: classifications(), id_classifications(), is_branch(), is_internode(), is_leaf(), is_root(), is_stem(), map_data(), map_data_(), n_leaves(), n_subtaxa(), n_subtaxa_1(), n_supertaxa(), n_supertaxa_1(), taxon_ids(), taxon_indexes(), taxon_names(), taxon_ranks()

Examples

```
# Get number of leaves for each taxon
n_leaves_1(ex_taxmap)

# Filter taxa based on number of leaves
filter_taxa(ex_taxmap, n_leaves_1 > 0)
```

`n_obs` *Count observations in [taxmap()]*

Description

Count observations for each taxon in a data set in a [taxmap()] object. This includes observations for the specific taxon and the observations of its subtaxa. "Observations" in this sense are the items (for list/vectors) or rows (for tables) in a dataset. By default, observations in the first data set in the [taxmap()] object is used. For example, if the data set is a table, then a value of 3 for a taxon means that their are 3 rows in that table assigned to that taxon or one of its subtaxa.

```
obj$n_obs(data)
n_obs(obj, data)
```

Arguments

`obj`	([taxmap()])
`data`	Dataset name, index, or a logical vector that indicates which dataset in 'obj$data' to add columns to.
`target`	DEPRECIATED. use "data" instead.

Value

'numeric'

See Also

Other taxmap data functions: `n_obs_1()`

Examples

```
# Get number of observations for each taxon in first dataset
n_obs(ex_taxmap)

# Get number of observations in a specified data set
n_obs(ex_taxmap, "info")
n_obs(ex_taxmap, "abund")

# Filter taxa using number of observations in the first table
filter_taxa(ex_taxmap, n_obs > 1)
```

n_obs_1 *Count observation assigned in [taxmap()]*

Description

Count observations for each taxon in a data set in a [taxmap()] object. This includes observations for the specific taxon but NOT the observations of its subtaxa. "Observations" in this sense are the items (for list/vectors) or rows (for tables) in a dataset. By default, observations in the first data set in the [taxmap()] object is used. For example, if the data set is a table, then a value of 3 for a taxon means that their are 3 rows in that table assigned to that taxon.

```
obj$n_obs_1(data)
n_obs_1(obj, data)
```

Arguments

obj	([taxmap()])
data	Dataset name, index, or a logical vector that indicates which dataset in 'obj$data' to add columns to.
target	DEPRECIATED. use "data" instead.

Value

'numeric'

See Also

Other taxmap data functions: n_obs()

Examples

```
# Get number of observations for each taxon in first dataset
n_obs_1(ex_taxmap)

# Get number of observations in a specified data set
n_obs_1(ex_taxmap, "info")
n_obs_1(ex_taxmap, "abund")

# Filter taxa using number of observations in the first table
filter_taxa(ex_taxmap, n_obs_1 > 0)
```

n_subtaxa *Get number of subtaxa*

Description

Get number of subtaxa for each taxon in an object of type [taxonomy()] or [taxmap()]

```
obj$n_subtaxa()
n_subtaxa(obj)
```

Arguments

obj	([taxonomy()] or [taxmap()])

Value

numeric

See Also

Other taxonomy data functions: classifications(), id_classifications(), is_branch(), is_internode(), is_leaf(), is_root(), is_stem(), map_data(), map_data_(), n_leaves(), n_leaves_1(), n_subtaxa_1(), n_supertaxa(), n_supertaxa_1(), taxon_ids(), taxon_indexes(), taxon_names(), taxon_ranks()

Examples

```
# Count number of subtaxa within each taxon
n_subtaxa(ex_taxmap)

# Filter taxa based on number of subtaxa
#  (this command removed all leaves or "tips" of the tree)
filter_taxa(ex_taxmap, n_subtaxa > 0)
```

n_subtaxa_1 *Get number of subtaxa*

Description

Get number of subtaxa for each taxon in an object of type [taxonomy()] or [taxmap()], not including subtaxa of subtaxa etc. This does not include subtaxa assigned to subtaxa.

```
obj$n_subtaxa_1()
n_subtaxa_1(obj)
```

Arguments

obj	([taxonomy()] or [taxmap()])

Value

numeric

See Also

Other taxonomy data functions: classifications(), id_classifications(), is_branch(), is_internode(), is_leaf(), is_root(), is_stem(), map_data(), map_data_(), n_leaves(), n_leaves_1(), n_subtaxa(), n_supertaxa(), n_supertaxa_1(), taxon_ids(), taxon_indexes(), taxon_names(), taxon_ranks()

Examples

```
# Count number of immediate subtaxa in each taxon
n_subtaxa_1(ex_taxmap)

# Filter taxa based on number of subtaxa
#  (this command removed all leaves or "tips" of the tree)
filter_taxa(ex_taxmap, n_subtaxa_1 > 0)
```

---

n_supertaxa *Get number of supertaxa*

Description

Get number of supertaxa for each taxon in an object of type [taxonomy()] or [taxmap()].

```
obj$n_supertaxa()
n_supertaxa(obj)
```

Arguments

obj	([taxonomy()] or [taxmap()])

Value

numeric

See Also

Other taxonomy data functions: classifications(), id_classifications(), is_branch(), is_internode(), is_leaf(), is_root(), is_stem(), map_data(), map_data_(), n_leaves(), n_leaves_1(), n_subtaxa(), n_subtaxa_1(), n_supertaxa_1(), taxon_ids(), taxon_indexes(), taxon_names(), taxon_ranks()

Examples

```
# Count number of supertaxa that contain each taxon
n_supertaxa(ex_taxmap)

# Filter taxa based on the number of supertaxa
#  (this command removes all root taxa)
filter_taxa(ex_taxmap, n_supertaxa > 0)
```

`n_supertaxa_1` *Get number of supertaxa*

Description

Get number of immediate supertaxa (i.e. not supertaxa of supertaxa, etc) for each taxon in an object of type [taxonomy()] or [taxmap()]. This should always be either 1 or 0.

```
obj$n_supertaxa_1()
n_supertaxa_1(obj)
```

Arguments

`obj`	([taxonomy()] or [taxmap()])

Value

`numeric`

See Also

Other taxonomy data functions: `classifications()`, `id_classifications()`, `is_branch()`, `is_internode()`, `is_leaf()`, `is_root()`, `is_stem()`, `map_data()`, `map_data_()`, `n_leaves()`, `n_leaves_1()`, `n_subtaxa()`, `n_subtaxa_1()`, `n_supertaxa()`, `taxon_ids()`, `taxon_indexes()`, `taxon_names()`, `taxon_ranks()`

Examples

```
# Test for the presence of supertaxa containing each taxon
n_supertaxa_1(ex_taxmap)

# Filter taxa based on the presence of supertaxa
#  (this command removes all root taxa)
filter_taxa(ex_taxmap, n_supertaxa_1 > 0)
```

obs *Get data indexes associated with taxa*

Description

Given a [taxmap()] object, return data associated with each taxon in a given table included in that [taxmap()] object.

```
obj$obs(data, value = NULL, subset = NULL,
  recursive = TRUE, simplify = FALSE)
obs(obj, data, value = NULL, subset = NULL,
  recursive = TRUE, simplify = FALSE)
```

Arguments

obj	([taxmap()]) The [taxmap()] object containing taxon information to be queried.
data	Either the name of something in 'obj$data' that has taxon information or a an external object with taxon information. For tables, there must be a column named "taxon_id" and lists/vectors must be named by taxon ID.
value	What data to return. This is usually the name of column in a table in 'obj$data'. Any result of 'all_names(obj)' can be used. If the value used has names, it is assumed that the names are taxon ids and the taxon ids are used to look up the correct values.
subset	Taxon IDs, TRUE/FALSE vector, or taxon indexes to find observations for. Default: All taxa in 'obj' will be used. Any variable name that appears in [all_names()] can be used as if it was a vector on its own.
recursive	('logical' or 'numeric') If 'FALSE', only return the observation assigned to the specified input taxa, not subtaxa. If 'TRUE', return all the observations of every subtaxa, etc. Positive numbers indicate the number of ranks below the each taxon to get observations for '0' is equivalent to 'FALSE'. Negative numbers are equivalent to 'TRUE'.
simplify	('logical') If 'TRUE', then combine all the results into a single vector of unique observation indexes.

Value

If 'simplify = FALSE', then a list of vectors of observation indexes are returned corresponding to the 'data' argument. If 'simplify = TRUE', then the observation indexes for all 'data' taxa are returned in a single vector.

Examples

```
# Get indexes of rows corresponding to each taxon
obs(ex_taxmap, "info")

# Get only a subset of taxon indexes
```

```
obs(ex_taxmap, "info", subset = 1:2)

# Get only a subset of taxon IDs
obs(ex_taxmap, "info", subset = c("b", "c"))

# Get only a subset of taxa using logical tests
obs(ex_taxmap, "info", subset = taxon_ranks == "genus")

# Only return indexes of rows assinged to each taxon explicitly
obs(ex_taxmap, "info", recursive = FALSE)

# Lump all row indexes in a single vector
obs(ex_taxmap, "info", simplify = TRUE)

# Return values from a dataset instead of indexes
obs(ex_taxmap, "info", value = "name")
```

---

obs_apply *Apply function to observations per taxon*

Description

Apply a function to data for the observations for each taxon. This is similar to using [obs()] with [lapply()] or [sapply()].

```
obj$obs_apply(data, func, simplify = FALSE, value = NULL,
  subset = NULL, recursive = TRUE, ...)
obs_apply(obj, data, func, simplify = FALSE, value = NULL,
  subset = NULL, recursive = TRUE, ...)
```

Arguments

obj	The [taxmap()] object containing taxon information to be queried.
data	Either the name of something in 'obj$data' that has taxon information or a an external object with taxon information. For tables, there must be a column named "taxon_id" and lists/vectors must be named by taxon ID.
func	('function') The function to apply.
simplify	('logical') If 'TRUE', convert lists to vectors.
value	What data to give to the function. This is usually the name of column in a table in 'obj$data'. Any result of 'all_names(obj)' can be used, but it usually only makes sense to use columns in the dataset specified by the 'data' option. By default, the indexes of observation in 'data' are returned.
subset	Taxon IDs, TRUE/FALSE vector, or taxon indexes to use. Default: All taxa in 'obj' will be used. Any variable name that appears in [all_names()] can be used as if it was a vector on its own.

recursive	(‘logical‘ or ‘numeric‘) If ‘FALSE‘, only return the observation assigned to the specified input taxa, not subtaxa. If ‘TRUE‘, return all the observations of every subtaxa, etc. Positive numbers indicate the number of ranks below the each taxon to get observations for ‘0‘ is equivalent to ‘FALSE‘. Negative numbers are equivalent to ‘TRUE‘.
...	Extra arguments are passed to the function.

Examples

```
# Find the average number of legs in each taxon
obs_apply(ex_taxmap, "info", mean, value = "n_legs", simplify = TRUE)

# One way to implement `n_obs` and find the number of observations per taxon
obs_apply(ex_taxmap, "info", length, simplify = TRUE)
```

---

parse_dada2 *Convert the output of dada2 to a taxmap object*

Description

Convert the ASV table and taxonomy table returned by dada2 into a taxmap object. An example of the input format can be found by following the dada2 tutorial here: shttps://benjjneb.github.io/dada2/tutorial.html

Usage

```
parse_dada2(
  seq_table,
  tax_table,
  class_key = "taxon_name",
  class_regex = "(.*)",
  include_match = TRUE
)
```

Arguments

seq_table	The ASV abundance matrix, with rows as samples and columns as ASV ids or sequences
tax_table	The table with taxonomic classifications for ASVs, with ASVs in rows and taxonomic ranks as columns.
class_key	(‘character‘ of length 1) The identity of the capturing groups defined using ‘class_regex‘. The length of ‘class_key‘ must be equal to the number of capturing groups specified in ‘class_regex‘. Any names added to the terms will be used as column names in the output. At least one ‘"taxon_name"‘ must be specified. Only ‘"info"‘ can be used multiple times. Each term must be one of those described below: * ‘taxon_name‘: The name of a taxon. Not necessarily unique, but are interpretable by a particular ‘database‘. Requires an internet connection.

* ‘taxon_rank‘: The rank of the taxon. This will be used to add rank info into the output object that can be accessed by ‘out$taxon_ranks()‘. * ‘info‘: Arbitrary taxon info you want included in the output. Can be used more than once.

`class_regex` (‘character‘ of length 1) A regular expression with capturing groups indicating the locations of data for each taxon in the ‘class‘ term in the ‘key‘ argument. The identity of the information must be specified using the ‘class_key‘ argument. The ‘class_sep‘ option can be used to split the classification into data for each taxon before matching. If ‘class_sep‘ is ‘NULL‘, each match of ‘class_regex‘ defines a taxon in the classification.

`include_match` (‘logical‘ of length 1) If ‘TRUE‘, include the part of the input matched by ‘class_regex‘ in the output object.

Value

`taxmap`

See Also

Other parsers: `extract_tax_data()`, `lookup_tax_data()`, `parse_edge_list()`, `parse_greengenes()`, `parse_mothur_tax_summary()`, `parse_mothur_taxonomy()`, `parse_newick()`, `parse_phylo()`, `parse_phyloseq()`, `parse_qiime_biom()`, `parse_rdp()`, `parse_silva_fasta()`, `parse_tax_data()`, `parse_ubiome()`, `parse_unite_general()`

`parse_greengenes` *Parse Greengenes release*

Description

Parses the greengenes database.

Usage

```
parse_greengenes(tax_file, seq_file = NULL)
```

Arguments

`tax_file` (`character` of length 1) The file path to the greengenes taxonomy file.

`seq_file` (`character` of length 1) The file path to the greengenes sequence fasta file. This is optional.

Details

The taxonomy input file has a format like:

```
228054  k__Bacteria; p__Cyanobacteria; c__Synechococcophycideae; o__Synech...
844608  k__Bacteria; p__Cyanobacteria; c__Synechococcophycideae; o__Synech...
...
```

The optional sequence file has a format like:

```
>1111886
AACGAACGCTGGCGGCATGCCTAACACATGCAAGTCGAACGAGACCTTCGGGTCTAGTGGCGCACGGGTGCGTA...
>1111885
AGAGTTTGATCCTGGCTCAGAATGAACGCTGGCGGCGTGCCTAACACATGCAAGTCGTACGAGAAATCCCGAGC...
...
```

Value

taxmap

See Also

Other parsers: extract_tax_data(), lookup_tax_data(), parse_dada2(), parse_edge_list(), parse_mothur_tax_summary(), parse_mothur_taxonomy(), parse_newick(), parse_phylo(), parse_phyloseq(), parse_qiime_biom(), parse_rdp(), parse_silva_fasta(), parse_tax_data(), parse_ubiome(), parse_unite_general()

---

parse_mothur_taxonomy *Parse mothur Classify.seqs *.taxonomy output*

Description

Parse the '*.taxonomy' file that is returned by the 'Classify.seqs' command in mothur. If confidence scores are present, they are included in the output.

Usage

```
parse_mothur_taxonomy(file = NULL, text = NULL)
```

Arguments

file	(character of length 1) The file path to the input file. Either "file" or "text" must be used, but not both.
text	(character) An alternate input to "file". The contents of the file as a character. Either "file" or "text" must be used, but not both.

Details

The input file has a format like:

```
AY457915 Bacteria(100);Firmicutes(99);Clostridiales(99);Johnsone...
AY457914 Bacteria(100);Firmicutes(100);Clostridiales(100);Johnso...
AY457913 Bacteria(100);Firmicutes(100);Clostridiales(100);Johnso...
AY457912 Bacteria(100);Firmicutes(99);Clostridiales(99);Johnsone...
AY457911 Bacteria(100);Firmicutes(99);Clostridiales(98);Ruminoco...
```

or...

```
AY457915 Bacteria;Firmicutes;Clostridiales;Johnsonella_et_rel.;J...
AY457914 Bacteria;Firmicutes;Clostridiales;Johnsonella_et_rel.;J...
AY457913 Bacteria;Firmicutes;Clostridiales;Johnsonella_et_rel.;J...
AY457912 Bacteria;Firmicutes;Clostridiales;Johnsonella_et_rel.;J...
AY457911 Bacteria;Firmicutes;Clostridiales;Ruminococcus_et_rel.;...
```

Value

taxmap

See Also

Other parsers: extract_tax_data(), lookup_tax_data(), parse_dada2(), parse_edge_list(), parse_greengenes(), parse_mothur_tax_summary(), parse_newick(), parse_phylo(), parse_phyloseq(), parse_qiime_biom(), parse_rdp(), parse_silva_fasta(), parse_tax_data(), parse_ubiome(), parse_unite_general()

---

parse_mothur_tax_summary

*Parse mothur *.tax.summary Classify.seqs output*

Description

Parse the '*.tax.summary' file that is returned by the 'Classify.seqs' command in mothur.

Usage

```
parse_mothur_tax_summary(file = NULL, text = NULL, table = NULL)
```

Arguments

file	(character of length 1) The file path to the input file. Either "file", "text", or "table" must be used, but only one.
text	(character) An alternate input to "file". The contents of the file as a character. Either "file", "text", or "table" must be used, but only one.
table	(character of length 1) An already parsed data.frame or tibble. Either "file", "text", or "table" must be used, but only one.

Details

The input file has a format like:

```
taxlevel  rankID  taxon  daughterlevels  total A B C
0 0 Root 2 242 84 84 74
1 0.1 Bacteria 50 242 84 84 74
2 0.1.2 Actinobacteria 38 13 0 13 0
3 0.1.2.3 Actinomycetaceae-Bifidobacteriaceae 10 13 0 13 0
4 0.1.2.3.7 Bifidobacteriaceae 6 13 0 13 0
5 0.1.2.3.7.2 Bifidobacterium_choerinum_et_rel. 8 13 0 13 0
6 0.1.2.3.7.2.1 Bifidobacterium_angulatum_et_rel. 1 11 0 11 0
7 0.1.2.3.7.2.1.1 unclassified 1 11 0 11 0
8 0.1.2.3.7.2.1.1.1 unclassified 1 11 0 11 0
9 0.1.2.3.7.2.1.1.1.1 unclassified 1 11 0 11 0
10 0.1.2.3.7.2.1.1.1.1.1 unclassified 1 11 0 11 0
11 0.1.2.3.7.2.1.1.1.1.1.1 unclassified 1 11 0 11 0
12 0.1.2.3.7.2.1.1.1.1.1.1.1 unclassified 1 11 0 11 0
6 0.1.2.3.7.2.5 Bifidobacterium_longum_et_rel. 1 2 0 2 0
7 0.1.2.3.7.2.5.1 unclassified 1 2 0 2 0
8 0.1.2.3.7.2.5.1.1 unclassified 1 2 0 2 0
9 0.1.2.3.7.2.5.1.1.1 unclassified 1 2 0 2 0
```

or

```
taxon total A B C
"k__Bacteria";"p__Actinobacteria";"c__Actinobacteria";... 1 0 1 0
"k__Bacteria";"p__Actinobacteria";"c__Actinobacteria";... 1 0 1 0
"k__Bacteria";"p__Actinobacteria";"c__Actinobacteria";... 1 0 1 0
```

Value

taxmap

See Also

Other parsers: extract_tax_data(), lookup_tax_data(), parse_dada2(), parse_edge_list(), parse_greengenes(), parse_mothur_taxonomy(), parse_newick(), parse_phylo(), parse_phyloseq(), parse_qiime_biom(), parse_rdp(), parse_silva_fasta(), parse_tax_data(), parse_ubiome(), parse_unite_general()

---

parse_newick *Parse a Newick file*

Description

Parse a Newick file into a taxmap object.

Usage

```
parse_newick(file = NULL, text = NULL)
```

Arguments

file	(character of length 1) The file path to the input file. Either file or text must be supplied but not both.
text	(character of length 1) The raw text to parse. Either file or text must be supplied but not both.

Details

The input file has a format like:

```
(ant:17, (bat:31, cow:22):7, dog:22, (elk:33, fox:12):40);
(dog:20, (elephant:30, horse:60):20):50;
```

Value

taxmap

See Also

Other parsers: extract_tax_data(), lookup_tax_data(), parse_dada2(), parse_edge_list(), parse_greengenes(), parse_mothur_tax_summary(), parse_mothur_taxonomy(), parse_phylo(), parse_phyloseq(), parse_qiime_biom(), parse_rdp(), parse_silva_fasta(), parse_tax_data(), parse_ubiome(), parse_unite_general()

---

parse_phylo *Parse a phylo object*

Description

Parses a phylo object from the ape package.

Usage

```
parse_phylo(obj)
```

Arguments

obj	A phylo object from the ape package.

Value

taxmap

See Also

Other parsers: extract_tax_data(), lookup_tax_data(), parse_dada2(), parse_edge_list(), parse_greengenes(), parse_mothur_tax_summary(), parse_mothur_taxonomy(), parse_newick(), parse_phyloseq(), parse_qiime_biom(), parse_rdp(), parse_silva_fasta(), parse_tax_data(), parse_ubiome(), parse_unite_general()

parse_phyloseq *Convert a phyloseq to taxmap*

Description

Converts a phyloseq object to a taxmap object.

Usage

```
parse_phyloseq(obj, class_regex = "(.*)", class_key = "taxon_name")
```

Arguments

obj	A phyloseq object
class_regex	A regular expression used to parse data in the taxon names. There must be a capture group (a pair of parentheses) for each item in class_key. See parse_tax_data for examples of how this works.
class_key	(‘character‘ of length 1) The identity of the capturing groups defined using ‘class_regex‘. The length of ‘class_key‘ must be equal to the number of capturing groups specified in ‘class_regex‘. Any names added to the terms will be used as column names in the output. At least one ‘"taxon_name"‘ must be specified. Only ‘"info"‘ can be used multiple times. Each term must be one of those described below: * ‘taxon_name‘: The name of a taxon. Not necessarily unique, but are interpretable by a particular ‘database‘. Requires an internet connection. * ‘taxon_rank‘: The rank of the taxon. This will be used to add rank info into the output object that can be accessed by ‘out$taxon_ranks()‘. * ‘info‘: Arbitrary taxon info you want included in the output. Can be used more than once.

Value

A taxmap object

See Also

Other parsers: extract_tax_data(), lookup_tax_data(), parse_dada2(), parse_edge_list(), parse_greengenes(), parse_mothur_tax_summary(), parse_mothur_taxonomy(), parse_newick(), parse_phylo(), parse_qiime_biom(), parse_rdp(), parse_silva_fasta(), parse_tax_data(), parse_ubiome(), parse_unite_general()

Examples

```
if (requireNamespace("phyloseq", quietly = TRUE)) {
  # Parse example dataset
  library(phyloseq)
  data(GlobalPatterns)
  x <- parse_phyloseq(GlobalPatterns)
```

```
  # Plot data
  heat_tree(x,
            node_size = n_obs,
            node_color = n_obs,
            node_label = taxon_names,
            tree_label = taxon_names)
}
```

`parse_primersearch` *Parse EMBOSS primersearch output*

Description

Parses the output file from EMBOSS primersearch into a data.frame with rows corresponding to predicted amplicons and their associated information.

Usage

```
parse_primersearch(file_path)
```

Arguments

`file_path`	The path to a primersearch output file.

Value

A data frame with each row corresponding to amplicon data

See Also

`run_primersearch`

`parse_qiime_biom` *Parse a BIOM output from QIIME*

Description

Parses a file in BIOM format from QIIME into a taxmap object. This also seems to work with files from MEGAN. I have not tested if it works with other BIOM files.

Usage

```
parse_qiime_biom(file, class_regex = "(.*)", class_key = "taxon_name")
```

Arguments

file	(character of length 1) The file path to the input file.
class_regex	A regular expression used to parse data in the taxon names. There must be a capture group (a pair of parentheses) for each item in class_key. See parse_tax_data for examples of how this works.
class_key	(‘character‘ of length 1) The identity of the capturing groups defined using ‘class_regex‘. The length of ‘class_key‘ must be equal to the number of capturing groups specified in ‘class_regex‘. Any names added to the terms will be used as column names in the output. At least one ‘"taxon_name"‘ must be specified. Only ‘"info"‘ can be used multiple times. Each term must be one of those described below: * ‘taxon_name‘: The name of a taxon. Not necessarily unique, but are interpretable by a particular ‘database‘. Requires an internet connection. * ‘taxon_rank‘: The rank of the taxon. This will be used to add rank info into the output object that can be accessed by ‘out$taxon_ranks()‘. * ‘info‘: Arbitrary taxon info you want included in the output. Can be used more than once.

Details

This function was inspired by the tutorial created by Geoffrey Zahn at http://geoffreyzahn.com/getting-your-otu-table-into-r/.

Value

A taxmap object

See Also

Other parsers: extract_tax_data(), lookup_tax_data(), parse_dada2(), parse_edge_list(), parse_greengenes(), parse_mothur_tax_summary(), parse_mothur_taxonomy(), parse_newick(), parse_phylo(), parse_phyloseq(), parse_rdp(), parse_silva_fasta(), parse_tax_data(), parse_ubiome(), parse_unite_general()

---

parse_rdp *Parse RDP FASTA release*

Description

Parses an RDP reference FASTA file.

Usage

```
parse_rdp(input = NULL, file = NULL, include_seqs = TRUE, add_species = FALSE)
```

Arguments

input	(character) One of the following: **A character vector of sequences** See the example below for what this looks like. The parser read_fasta produces output like this. **A list of character vectors** Each vector should have one base per element. **A "DNAbin" object** This is the result of parsers like read.FASTA. **A list of "SeqFastadna" objects** This is the result of parsers like read.fasta. Either "input" or "file" must be supplied but not both.
file	The path to a FASTA file containing sequences to use. Either "input" or "file" must be supplied but not both.
include_seqs	(logical of length 1) If TRUE, include sequences in the output object.
add_species	(logical of length 1) If TRUE, add the species information to the taxonomy. In this database, the species name often contains other information as well.

Details

The input file has a format like:

```
>S000448483 Sparassis crispa; MBUH-PIRJO&ILKKA94-1587/ss5 Lineage=Root;rootrank;Fun...
ggattcccctagtaactgcgagtgaagcgggaagagctcaaatttaaaatctggcggcgtcctcgtcgtccgagttgtaa
tctggagaagcgacatccgcgctggaccgtgtacaagtctcttggaaaagagcgtcgtagagggtgacaatcccgtcttt
...
```

Value

taxmap

See Also

Other parsers: extract_tax_data(), lookup_tax_data(), parse_dada2(), parse_edge_list(), parse_greengenes(), parse_mothur_tax_summary(), parse_mothur_taxonomy(), parse_newick(), parse_phylo(), parse_phyloseq(), parse_qiime_biom(), parse_silva_fasta(), parse_tax_data(), parse_ubiome(), parse_unite_general()

---

parse_silva_fasta *Parse SILVA FASTA release*

Description

Parses an SILVA FASTA file that can be found at https://www.arb-silva.de/no_cache/download/archive/release_128/Exports/.

Usage

```
parse_silva_fasta(file = NULL, input = NULL, include_seqs = TRUE)
```

Arguments

file	The path to a FASTA file containing sequences to use. Either "input" or "file" must be supplied but not both.
input	(character) One of the following: **A character vector of sequences** See the example below for what this looks like. The parser read_fasta produces output like this. **A list of character vectors** Each vector should have one base per element. **A "DNAbin" object** This is the result of parsers like read.FASTA. **A list of "SeqFastadna" objects** This is the result of parsers like read.fasta. Either "input" or "file" must be supplied but not both.
include_seqs	(logical of length 1) If TRUE, include sequences in the output object.

Details

The input file has a format like:

```
 >GCVF01000431.1.2369
Bacteria;Proteobacteria;Gammaproteobacteria;Oceanospiril...
CGUGCACGGUGGAUGCCUUGGCAGCCAGAGGCGAUGAAGGACGUUGUAGCCUGCGAUAAGCUCCGGUUAGGUGGCAAACA
ACCGUUUGACCCGGAGAUCUCCGAAUGGGGCAACCCACCCGUUGUAAGGCGGGUAUCACCGACUGAAUCCAUAGGUCGGU
...
```

Value

taxmap

See Also

Other parsers: extract_tax_data(), lookup_tax_data(), parse_dada2(), parse_edge_list(), parse_greengenes(), parse_mothur_tax_summary(), parse_mothur_taxonomy(), parse_newick(), parse_phylo(), parse_phyloseq(), parse_qiime_biom(), parse_rdp(), parse_tax_data(), parse_ubiome(), parse_unite_general()

---

parse_tax_data *Convert one or more data sets to taxmap*

Description

Reads taxonomic information and associated data in tables, lists, and vectors and stores it in a [taxmap()] object. [Taxonomic classifications](https://en.wikipedia.org/wiki/Taxonomy_(biology)#Classifying_organisms) must be present.

Usage

```
parse_tax_data(
  tax_data,
  datasets = list(),
  class_cols = 1,
  class_sep = ";",
  sep_is_regex = FALSE,
  class_key = "taxon_name",
  class_regex = "(.*)",
  class_reversed = FALSE,
  include_match = TRUE,
  mappings = c(),
  include_tax_data = TRUE,
  named_by_rank = FALSE
)
```

Arguments

tax_data	A table, list, or vector that contains the names of taxa that represent [taxonomic classifications](https://en.wikipedia.org/wiki/Taxonomy_(biology)#Classifying_organisms). Accepted representations of classifications include: * A list/vector or table with column(s) of taxon names: Something like '"Animalia;Chordata;Mammalia;Primates;Hominidae;Homo" What separator(s) is used (";" in this example) can be changed with the 'class_sep' option. For tables, the classification can be spread over multiple columns and the separator(s) will be applied to each column, although each column could just be single taxon names with no separator. Use the 'class_cols' option to specify which columns have taxon names. * A list in which each entry is a classifications. For example, 'list(c("Animalia", "Chordata", "Mammalia", "Primates", "Hominidae", "Homo"), ...)'. * A list of data.frames where each represents a classification with one taxon per row. The column that contains taxon names is specified using the 'class_cols' option. In this instance, it only makes sense to specify a single column.
datasets	Additional lists/vectors/tables that should be included in the resulting 'taxmap' object. The 'mappings' option is use to specify how these data sets relate to the 'tax_data' and, by inference, what taxa apply to each item.
class_cols	('character' or 'integer') The names or indexes of columns that contain classifications if the first input is a table. If multiple columns are specified, they will be combined in the order given. Negative column indexes mean "every column besides these columns".
class_sep	('character') One or more separators that delineate taxon names in a classification. For example, if one column had '"Homo sapiens"' and another had '"Animalia;Chordata;Mammalia;Primates;Hominidae"', then 'class_sep = c(" ", ";")'. All separators are applied to each column so order does not matter.
sep_is_regex	('TRUE'/'FALSE') Whether or not 'class_sep' should be used as a [regular expression](https://en.wikipedia.org/wiki/Regular_expression).
class_key	('character' of length 1) The identity of the capturing groups defined using 'class_regex'. The length of 'class_key' must be equal to the number of cap-

turing groups specified in ‘class_regex‘. Any names added to the terms will be used as column names in the output. At least one ‘"taxon_name"‘ must be specified. Only ‘"info"‘ can be used multiple times. Each term must be one of those described below: * ‘taxon_name‘: The name of a taxon. Not necessarily unique, but are interpretable by a particular ‘database‘. Requires an internet connection. * ‘taxon_rank‘: The rank of the taxon. This will be used to add rank info into the output object that can be accessed by ‘out$taxon_ranks()‘. * ‘info‘: Arbitrary taxon info you want included in the output. Can be used more than once.

`class_regex` (‘character‘ of length 1) A regular expression with capturing groups indicating the locations of data for each taxon in the ‘class‘ term in the ‘key‘ argument. The identity of the information must be specified using the ‘class_key‘ argument. The ‘class_sep‘ option can be used to split the classification into data for each taxon before matching. If ‘class_sep‘ is ‘NULL‘, each match of ‘class_regex‘ defines a taxon in the classification.

`class_reversed` If ‘TRUE‘, then classifications go from specific to general. For example: ‘Abditomys latidens : Muridae : Rodentia : Mammalia : Chordata‘.

`include_match` (‘logical‘ of length 1) If ‘TRUE‘, include the part of the input matched by ‘class_regex‘ in the output object.

`mappings` (named ‘character‘) This defines how the taxonomic information in ‘tax_data‘ applies to data set in ‘datasets‘. This option should have the same number of inputs as ‘datasets‘, with values corresponding to each data set. The names of the character vector specify what information in ‘tax_data‘ is shared with info in each ‘dataset‘, which is specified by the corresponding values of the character vector. If there are no shared variables, you can add ‘NA‘ as a placeholder, but you could just leave that data out since it is not benefiting from being in the taxmap object. The names/values can be one of the following: * For tables, the names of columns can be used. * ‘"{{index}}"‘ : This means to use the index of rows/items * ‘"{{name}}"‘ : This means to use row/item names. * ‘"{{value}}"‘ : This means to use the values in vectors or lists. Lists will be converted to vectors using [unlist()].

`include_tax_data` (‘TRUE‘/‘FALSE‘) Whether or not to include ‘tax_data‘ as a dataset, like those in ‘datasets‘.

`named_by_rank` (‘TRUE‘/‘FALSE‘) If ‘TRUE‘ and the input is a table with columns named by ranks or a list of vectors with each vector named by ranks, include that rank info in the output object, so it can be accessed by ‘out$taxon_ranks()‘. If ‘TRUE‘, taxa with different ranks, but the same name and location in the taxonomy, will be considered different taxa. Cannot be used with the ‘sep‘, ‘class_regex‘, or ‘class_key‘ options.

See Also

Other parsers: `extract_tax_data()`, `lookup_tax_data()`, `parse_dada2()`, `parse_edge_list()`, `parse_greengenes()`, `parse_mothur_tax_summary()`, `parse_mothur_taxonomy()`, `parse_newick()`, `parse_phylo()`, `parse_phyloseq()`, `parse_qiime_biom()`, `parse_rdp()`, `parse_silva_fasta()`, `parse_ubiome()`, `parse_unite_general()`

Examples

```
# Read a vector of classifications
my_taxa <- c("Mammalia;Carnivora;Felidae",
             "Mammalia;Carnivora;Felidae",
             "Mammalia;Carnivora;Ursidae")
parse_tax_data(my_taxa, class_sep = ";")

# Read a list of classifications
my_taxa <- list("Mammalia;Carnivora;Felidae",
               "Mammalia;Carnivora;Felidae",
               "Mammalia;Carnivora;Ursidae")
parse_tax_data(my_taxa, class_sep = ";")

# Read classifications in a table in a single column
species_data <- data.frame(tax = c("Mammalia;Carnivora;Felidae",
                                   "Mammalia;Carnivora;Felidae",
                                   "Mammalia;Carnivora;Ursidae"),
                           species_id = c("A", "B", "C"))
parse_tax_data(species_data, class_sep = ";", class_cols = "tax")

# Read classifications in a table in multiple columns
species_data <- data.frame(lineage = c("Mammalia;Carnivora;Felidae",
                                       "Mammalia;Carnivora;Felidae",
                                       "Mammalia;Carnivora;Ursidae"),
                           species = c("Panthera leo",
                                       "Panthera tigris",
                                       "Ursus americanus"),
                           species_id = c("A", "B", "C"))
parse_tax_data(species_data, class_sep = c(" ", ";"),
               class_cols = c("lineage", "species"))

# Read classification tables with one column per rank
species_data <- data.frame(class = c("Mammalia", "Mammalia", "Mammalia"),
                           order = c("Carnivora", "Carnivora", "Carnivora"),
                           family = c("Felidae", "Felidae", "Ursidae"),
                           genus = c("Panthera", "Panthera", "Ursus"),
                           species = c("leo", "tigris", "americanus"),
                           species_id = c("A", "B", "C"))
 parse_tax_data(species_data, class_cols = 1:5)
 parse_tax_data(species_data, class_cols = 1:5,
                named_by_rank = TRUE) # makes `taxon_ranks()` work

# Classifications with extra information
my_taxa <- c("Mammalia_class_1;Carnivora_order_2;Felidae_genus_3",
             "Mammalia_class_1;Carnivora_order_2;Felidae_genus_3",
             "Mammalia_class_1;Carnivora_order_2;Ursidae_genus_3")
parse_tax_data(my_taxa, class_sep = ";",
               class_regex = "(.+)_(.+)_([0-9]+)",
               class_key = c(my_name = "taxon_name",
                             a_rank = "taxon_rank",
                             some_num = "info"))
```

```
# --- Parsing multiple datasets at once (advanced) ---
# The rest is one example for how to classify multiple datasets at once.

# Make example data with taxonomic classifications
species_data <- data.frame(tax = c("Mammalia;Carnivora;Felidae",
                                   "Mammalia;Carnivora;Felidae",
                                   "Mammalia;Carnivora;Ursidae"),
                           species = c("Panthera leo",
                                       "Panthera tigris",
                                       "Ursus americanus"),
                           species_id = c("A", "B", "C"))

# Make example data associated with the taxonomic data
# Note how this does not contain classifications, but
# does have a varaible in common with "species_data" ("id" = "species_id")
abundance <- data.frame(id = c("A", "B", "C", "A", "B", "C"),
                        sample_id = c(1, 1, 1, 2, 2, 2),
                        counts = c(23, 4, 3, 34, 5, 13))

# Make another related data set named by species id
common_names <- c(A = "Lion", B = "Tiger", C = "Bear", "Oh my!")

# Make another related data set with no names
foods <- list(c("ungulates", "boar"),
              c("ungulates", "boar"),
              c("salmon", "fruit", "nuts"))

# Make a taxmap object with these three datasets
x = parse_tax_data(species_data,
                   datasets = list(counts = abundance,
                                   my_names = common_names,
                                   foods = foods),
                   mappings = c("species_id" = "id",
                                "species_id" = "{{name}}",
                                "{{index}}" = "{{index}}"),
                   class_cols = c("tax", "species"),
                   class_sep = c(" ", ";"))

# Note how all the datasets have taxon ids now
x$data

# This allows for complex mappings between variables that other functions use
map_data(x, my_names, foods)
map_data(x, counts, my_names)
```

---

parse_ubiome *Converts the uBiome file format to taxmap*

Description

Converts the uBiome file format to taxmap. NOTE: This is experimental and might not work if uBiome changes their format. Contact the maintainers if you encounter problems/

Usage

```
parse_ubiome(file = NULL, table = NULL)
```

Arguments

file	(character of length 1) The file path to the input file. Either "file", or "table" must be used, but only one.
table	(character of length 1) An already parsed data.frame or tibble. Either "file", or "table" must be used, but only one.

Details

The input file has a format like:

```
tax_name,tax_rank,count,count_norm,taxon,parent
root,root,29393,1011911,1,
Bacteria,superkingdom,29047,1000000,2,131567
Campylobacter,genus,23,791,194,72294
Flavobacterium,genus,264,9088,237,49546
```

Value

taxmap

See Also

Other parsers: extract_tax_data(), lookup_tax_data(), parse_dada2(), parse_edge_list(), parse_greengenes(), parse_mothur_tax_summary(), parse_mothur_taxonomy(), parse_newick(), parse_phylo(), parse_phyloseq(), parse_qiime_biom(), parse_rdp(), parse_silva_fasta(), parse_tax_data(), parse_unite_general()

---

parse_unite_general *Parse UNITE general release FASTA*

Description

Parse the UNITE general release FASTA file

Usage

```
parse_unite_general(input = NULL, file = NULL, include_seqs = TRUE)
```

Arguments

input	(character) One of the following:
	A character vector of sequences See the example below for what this looks like. The parser read_fasta produces output like this.
	A list of character vectors Each vector should have one base per element.
	A "DNAbin" object This is the result of parsers like read.FASTA.
	A list of "SeqFastadna" objects This is the result of parsers like read.fasta. Either "input" or "file" must be supplied but not both.
file	The path to a FASTA file containing sequences to use. Either "input" or "file" must be supplied but not both.
include_seqs	(logical of length 1) If TRUE, include sequences in the output object.

Details

The input file has a format like:

```
>Glomeromycota_sp|KJ484724|SH523877.07FU|reps|k__Fungi;p__Glomeromycota;c__unid...
ATAATTTGCCGAACCTAGCGTTAGCGCGAGGTTCTGCGATCAACACTTATATTTAAAACCCAACTCTTAAATTTTGTAT...
```

Value

taxmap

See Also

Other parsers: extract_tax_data(), lookup_tax_data(), parse_dada2(), parse_edge_list(), parse_greengenes(), parse_mothur_tax_summary(), parse_mothur_taxonomy(), parse_newick(), parse_phylo(), parse_phyloseq(), parse_qiime_biom(), parse_rdp(), parse_silva_fasta(), parse_tax_data(), parse_ubiome()

---

primersearch *Use EMBOSS primersearch for in silico PCR*

Description

A pair of primers are aligned against a set of sequences. A taxmap object with two tables is returned: a table with information for each predicted amplicon, quality of match, and predicted amplicons, and a table with per-taxon amplification statistics. Requires the EMBOSS tool kit (https://emboss.sourceforge.net/) to be installed.

Usage

```
primersearch(obj, seqs, forward, reverse, mismatch = 5, clone = TRUE)
```

Arguments

obj	A taxmap object.
seqs	The sequences to do in silico PCR on. This can be any variable in obj$data listed in all_names(obj) or an external variable. If an external variable (i.e. not in obj$data), it must be named by taxon IDs or have the same length as the number of taxa in obj. Currently, only character vectors are accepted.
forward	(character of length 1) The forward primer sequence
reverse	(character of length 1) The reverse primer sequence
mismatch	An integer vector of length 1. The percentage of mismatches allowed.
clone	If TRUE, make a copy of the input object and add on the results (like most R functions). If FALSE, the input will be changed without saving the result, which uses less RAM.

Details

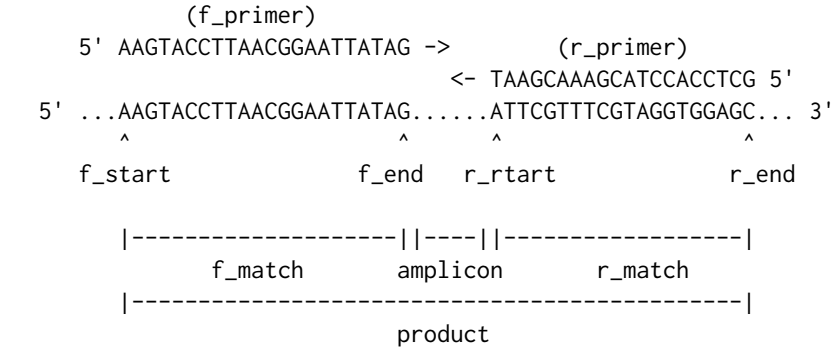
It can be confusing how the primer sequence relates to the binding sites on a reference database sequence. A simplified diagram can help. For example, if the top strand below (5' -> 3') is the database sequence, the forward primer has the same sequence as the target region, since it will bind to the other strand (3' -> 5') during PCR and extend on the 3' end. However, the reverse primer must bind to the database strand, so it will have to be the complement of the reference sequence. It also has to be reversed to make it in the standard 5' -> 3' orientation. Therefore, the reverse primer must be the reverse complement of its binding site on the reference sequence.

```
Primer 1: 5' AAGTACCTTAACGGAATTATAG 3'
Primer 2: 5' GCTCCACCTACGAAACGAAT   3'

                                <- TAAGCAAAGCATCCACCTCG 5'
5' ...AAGTACCTTAACGGAATTATAG......ATTCGTTTCGTAGGTGGAGC... 3'

3' ...TTCATGGAATTGCCTTAATATC......TAAGCAAAGCATCCACCTCG... 5'
   5' AAGTACCTTAACGGAATTATAG ->
```

However, a database might have either the top or the bottom strand as a reference sequence. Since one implies the sequence of the other, either is valid, but this is another source of confusion. If we take the diagram above and rotate it 180 degrees, it would mean the same thing, but which primer we would want to call "forward" and which we would want to call "reverse" would change. Databases of a single locus (e.g. Greengenes) will likely have a convention for which strand will be present, so relative to this convention, there is a distinct "forward" and "reverse". However, computers dont know about this convention, so the "forward" primer is whichever primer has the same sequence as its binding region in the database (as opposed to the reverse complement). For this reason, primersearch will redefine which primer is "forward" and which is "reverse" based on how it binds the reference sequence. See the example code in primersearch_raw for a demonstration of this.

Value

A copy of the input taxmap object with two tables added. One table contains amplicon information with one row per predicted amplicon with the following info:



taxon_id: The taxon IDs for the sequence.

seq_index: The index of the input sequence.

f_primer: The sequence of the forward primer.

r_primer: The sequence of the reverse primer.

f_mismatch: The number of mismatches on the forward primer.

r_mismatch: The number of mismatches on the reverse primer.

f_start: The start location of the forward primer.

f_end: The end location of the forward primer.

r_start: The start location of the reverse primer.

r_end: The end location of the reverse primer.

f_match: The sequence matched by the forward primer.

r_match: The sequence matched by the reverse primer.

amplicon: The sequence amplified by the primers, not including the primers.

product: The sequence amplified by the primers including the primers. This simulates a real PCR product.

The other table contains per-taxon information about the PCR, with one row per taxon. It has the following columns:

taxon_ids: Taxon IDs.

query_count: The number of sequences used as input.

seq_count: The number of sequences that had at least one amplicon.

amp_count: The number of amplicons. Might be more than one per sequence.

amplified: If at least one sequence of that taxon had at least one amplicon.

multiple: If at least one sequences had at least two amplicons.

prop_amplified: The proportion of sequences with at least one amplicon.

med_amp_len: The median amplicon length.

min_amp_len: The minimum amplicon length.

max_amp_len: The maximum amplicon length.

med_prod_len: The median product length.

min_prod_len: The minimum product length.

max_prod_len: The maximum product length.

Installing EMBOSS

The command-line tool "primersearch" from the EMBOSS tool kit is needed to use this function. How you install EMBOSS will depend on your operating system:

Linux:

Open up a terminal and type:

`sudo apt-get install emboss`

Mac OSX:

The easiest way to install EMBOSS on OSX is to use homebrew. After installing homebrew, open up a terminal and type:

`brew install homebrew/science/emboss`

Windows:

There is an installer for Windows here:

ftp://emboss.open-bio.org/pub/EMBOSS/windows/mEMBOSS-6.5.0.0-setup.exe

Examples

```
if (primersearch_is_installed()) {
  # Get example FASTA file
  fasta_path <- system.file(file.path("extdata", "silva_subset.fa"),
                            package = "metacoder")

  # Parse the FASTA file as a taxmap object
  obj <- parse_silva_fasta(file = fasta_path)

  # Simulate PCR with primersearch
  # Have to replace Us with Ts in sequences since primersearch
  #   does not understand Us.
  obj <- primersearch(obj,
                      gsub(silva_seq, pattern = "U", replace = "T"),
                      forward = c("U519F" = "CAGYMGCCRCGGKAAHACC"),
                      reverse = c("Arch806R" = "GGACTACNSGGGTMTCTAAT"),
                      mismatch = 10)

  # Plot what did not amplify
  obj %>%
    filter_taxa(prop_amplified < 1) %>%
    heat_tree(node_label = taxon_names,
              node_color = prop_amplified,
              node_color_range = c("grey", "red", "purple", "green"),
              node_color_trans = "linear",
              node_color_axis_label = "Proportion amplified",
              node_size = n_obs,
              node_size_axis_label = "Number of sequences",
              layout = "da",
              initial_layout = "re")
}
```

primersearch_is_installed *Test if primersearch is installed*

Description

Check if primersearch is installed

Usage

```
primersearch_is_installed(must_be_installed = FALSE)
```

Arguments

must_be_installed	(logical of length 1) If TRUE, throw an error if primersearch is not installed.

Value

logical of length 1

primersearch_raw *Use EMBOSS primersearch for in silico PCR*

Description

A pair of primers are aligned against a set of sequences. The location of the best hits, quality of match, and predicted amplicons are returned. Requires the EMBOSS tool kit (https://emboss.sourceforge.net/) to be installed.

Usage

```
primersearch_raw(input = NULL, file = NULL, forward, reverse, mismatch = 5)
```

Arguments

input	(character) One of the following: **A character vector of sequences** See the example below for what this looks like. The parser read_fasta produces output like this. **A list of character vectors** Each vector should have one base per element. **A "DNAbin" object** This is the result of parsers like read.FASTA. **A list of "SeqFastadna" objects** This is the result of parsers like read.fasta. Either "input" or "file" must be supplied but not both.
file	The path to a FASTA file containing sequences to use. Either "input" or "file" must be supplied but not both.

forward	(character of length 1) The forward primer sequence
reverse	(character of length 1) The reverse primer sequence
mismatch	An integer vector of length 1. The percentage of mismatches allowed.

Details

It can be confusing how the primer sequence relates to the binding sites on a reference database sequence. A simplified diagram can help. For example, if the top strand below (5' -> 3') is the database sequence, the forward primer has the same sequence as the target region, since it will bind to the other strand (3' -> 5') during PCR and extend on the 3' end. However, the reverse primer must bind to the database strand, so it will have to be the complement of the reference sequence. It also has to be reversed to make it in the standard 5' -> 3' orientation. Therefore, the reverse primer must be the reverse complement of its binding site on the reference sequence.

```
Primer 1: 5' AAGTACCTTAACGGAATTATAG 3'
Primer 2: 5' GCTCCACCTACGAAACGAAT   3'

                                <- TAAGCAAAGCATCCACCTCG 5'
5' ...AAGTACCTTAACGGAATTATAG......ATTCGTTTCGTAGGTGGAGC... 3'

3' ...TTCATGGAATTGCCTTAATATC......TAAGCAAAGCATCCACCTCG... 5'
   5' AAGTACCTTAACGGAATTATAG ->
```

However, a database might have either the top or the bottom strand as a reference sequence. Since one implies the sequence of the other, either is valid, but this is another source of confusion. If we take the diagram above and rotate it 180 degrees, it would mean the same thing, but which primer we would want to call "forward" and which we would want to call "reverse" would change. Databases of a single locus (e.g. Greengenes) will likely have a convention for which strand will be present, so relative to this convention, there is a distinct "forward" and "reverse". However, computers dont know about this convention, so the "forward" primer is whichever primer has the same sequence as its binding region in the database (as opposed to the reverse complement). For this reason, primersearch will redefine which primer is "forward" and which is "reverse" based on how it binds the reference sequence. See the example code for a demonstration of this.

Value

A table with one row per predicted amplicon with the following info:

```
          (f_primer)
   5' AAGTACCTTAACGGAATTATAG ->        (r_primer)
                                <- TAAGCAAAGCATCCACCTCG 5'
5' ...AAGTACCTTAACGGAATTATAG......ATTCGTTTCGTAGGTGGAGC... 3'
      ^                     ^      ^                   ^
   f_start              f_end   r_rtart             r_end

      |--------------------||----||------------------|
             f_match       amplicon       r_match
      |-----------------------------------------------|
```

product

f_mismatch: The number of mismatches on the forward primer
r_mismatch: The number of mismatches on the reverse primer
input: The index of the input sequence

Installing EMBOSS

The command-line tool "primersearch" from the EMBOSS tool kit is needed to use this function. How you install EMBOSS will depend on your operating system:

Linux:

Open up a terminal and type:

`sudo apt-get install emboss`

Mac OSX:

The easiest way to install EMBOSS on OSX is to use homebrew. After installing homebrew, open up a terminal and type:

`brew install homebrew/science/emboss`

Windows:

There is an installer for Windows here:

ftp://emboss.open-bio.org/pub/EMBOSS/windows/mEMBOSS-6.5.0.0-setup.exe

Examples

```
if (primersearch_is_installed()) {
  ### Dummy test data set ###

  primer_1_site <- "AAGTACCTTAACGGAATTATAG"
  primer_2_site <- "ATTCGTTTCGTAGGTGGAGC"
  amplicon <- "NNNAGTGGATAGATAGGGGTTCTGTGGCGTTTGGGAATTAAAGATTAGAGANNN"
  seq_1 <- paste0("AA", primer_1_site, amplicon, primer_2_site, "AAAA")
  seq_2 <- rev_comp(seq_1)
  f_primer <- "ACGTACCTTAACGGAATTATAG" # Note the "C" mismatch at position 2
  r_primer <- rev_comp(primer_2_site)
  seqs <- c(a = seq_1, b = seq_2)

  result <- primersearch_raw(seqs, forward = f_primer, reverse = r_primer)


  ### Real data set ###

  # Get example FASTA file
  fasta_path <- system.file(file.path("extdata", "silva_subset.fa"),
                            package = "metacoder")

  # Parse the FASTA file as a taxmap object
  obj <- parse_silva_fasta(file = fasta_path)
```

```
  # Simulate PCR with primersearch
  pcr_result <- primersearch_raw(obj$data$tax_data$silva_seq,
                                 forward = c("U519F" = "CAGYMGCCRCGGKAAHACC"),
                                 reverse = c("Arch806R" = "GGACTACNSGGGTMTCTAAT"),
                                 mismatch = 10)

  # Add result to input table
  #  NOTE: We want to add a function to handle running pcr on a
  #        taxmap object directly, but we are still trying to figure out
  #        the best way to implement it. For now, do the following:
  obj$data$pcr <- pcr_result
  obj$data$pcr$taxon_id <- obj$data$tax_data$taxon_id[pcr_result$input]

  # Visualize which taxa were amplified
  #  This work because only amplicons are returned by `primersearch`
  n_amplified <- unlist(obj$obs_apply("pcr",
      function(x) length(unique(obj$data$tax_data$input[x]))))
  prop_amped <- n_amplified / obj$n_obs()
  heat_tree(obj,
            node_label = taxon_names,
            node_color = prop_amped,
            node_color_range = c("grey", "red", "purple", "green"),
            node_color_trans = "linear",
            node_color_axis_label = "Proportion amplified",
            node_size = n_obs,
            node_size_axis_label = "Number of sequences",
            layout = "da",
            initial_layout = "re")
}
```

`print_tree` *Print a text tree*

Description

Print a text-based tree of a [taxonomy()] or [taxmap()] object.

Arguments

`obj`	A `taxonomy` or `taxmap` object
`value`	What data to return. Default is taxon names. Any result of [all_names()] can be used, but it usually only makes sense to use data with one value per taxon, like taxon names.

Examples

```
print_tree(ex_taxmap)
```

qualitative_palette *The default qualitative color palette*

Description

Returns the default color palette for qualitative data

Usage

```
qualitative_palette()
```

Value

`character` of hex color codes

Examples

```
qualitative_palette()
```

quantative_palette *The default quantative color palette*

Description

Returns the default color palette for quantative data.

Usage

```
quantative_palette()
```

Value

`character` of hex color codes

Examples

```
quantative_palette()
```

ranks_ref *Lookup-table for IDs of taxonomic ranks*

Description

Composed of two columns:

- rankid - the ordered identifier value. lower values mean higher rank
- ranks - all the rank names that belong to the same level, with different variants that mean essentially the same thing

rarefy_obs *Calculate rarefied observation counts*

Description

For a given table in a `taxmap` object, rarefy counts to a constant total. This is a wrapper around `rrarefy` that automatically detects which columns are numeric and handles the reformatting needed to use tibbles.

Usage

```
rarefy_obs(
  obj,
  data,
  sample_size = NULL,
  cols = NULL,
  other_cols = FALSE,
  out_names = NULL,
  dataset = NULL
)
```

Arguments

`obj`	A `taxmap` object
`data`	The name of a table in `obj$data`.
`sample_size`	The sample size counts will be rarefied to. This can be either a single integer or a vector of integers of equal length to the number of columns.
`cols`	The columns in `data` to use. By default, all numeric columns are used. Takes one of the following inputs: **TRUE/FALSE:** All/No columns will used. **Character vector:** The names of columns to use **Numeric vector:** The indexes of columns to use

Vector of TRUE/FALSE of length equal to the number of columns: Use the columns corresponding to TRUE values.

other_cols — Preserve in the output non-target columns present in the input data. New columns will always be on the end. The "taxon_id" column will be preserved in the front. Takes one of the following inputs:

NULL: No columns will be added back, not even the taxon id column.

TRUE/FALSE: All/None of the non-target columns will be preserved.

Character vector: The names of columns to preserve

Numeric vector: The indexes of columns to preserve

Vector of TRUE/FALSE of length equal to the number of columns: Preserve the columns corresponding to TRUE values.

out_names — The names of count columns in the output. Must be the same length and order as cols (or unique(groups), if groups is used).

dataset — DEPRECIATED. use "data" instead.

Value

A tibble

See Also

Other calculations: calc_diff_abund_deseq2(), calc_group_mean(), calc_group_median(), calc_group_rsd(), calc_group_stat(), calc_n_samples(), calc_obs_props(), calc_prop_samples(), calc_taxon_abund(), compare_groups(), counts_to_presence(), zero_low_counts()

Examples

```
# Parse data for examples
x = parse_tax_data(hmp_otus, class_cols = "lineage", class_sep = ";",
                   class_key = c(tax_rank = "taxon_rank", tax_name = "taxon_name"),
                   class_regex = "^(.+)__(.+)$")

# Rarefy all numeric columns
rarefy_obs(x, "tax_data")

# Rarefy a subset of columns
rarefy_obs(x, "tax_data", cols = c("700035949", "700097855", "700100489"))
rarefy_obs(x, "tax_data", cols = 4:6)
rarefy_obs(x, "tax_data", cols = startsWith(colnames(x$data$tax_data), "70001"))

# Including all other columns in ouput
rarefy_obs(x, "tax_data", other_cols = TRUE)

# Inlcuding specific columns in output
rarefy_obs(x, "tax_data", cols = c("700035949", "700097855", "700100489"),
               other_cols = 2:3)

# Rename output columns
rarefy_obs(x, "tax_data", cols = c("700035949", "700097855", "700100489"),
               out_names = c("a", "b", "c"))
```

read_fasta *Read a FASTA file*

Description

Reads a FASTA file. This is the FASTA parser for metacoder. It simply tries to read a FASTA file into a named character vector with minimal fuss. It does not do any checks for valid characters etc. Other FASTA parsers you might want to consider include `read.FASTA` or `read.fasta`.

Usage

```
read_fasta(file_path)
```

Arguments

`file_path`	(`character` of length 1) The path to a file to read.

Value

named `character` vector

Examples

```
# Get example FASTA file
fasta_path <- system.file(file.path("extdata", "silva_subset.fa"),
                          package = "metacoder")

# Read fasta file
my_seqs <- read_fasta(fasta_path)
```

remove_redundant_names *Remove redundant parts of taxon names*

Description

Remove the names of parent taxa in the beginning of their children's names in a `taxonomy` or `taxmap` object. This is useful for removing genus names in species binomials.

```
obj$remove_redundant_names()
remove_redundant_names(obj)
```

Arguments

`obj`	A `taxonomy` or `taxmap` object

Value

A taxonomy or taxmap object

Examples

```
# Remove genus named from species taxa
species_data <- c("Carnivora;Felidae;Panthera;Panthera leo",
                  "Carnivora;Felidae;Panthera;Panthera tigris",
                  "Carnivora;Ursidae;Ursus;Ursus americanus")
obj <-  parse_tax_data(species_data, class_sep = ";")
remove_redundant_names(obj)
```

---

replace_taxon_ids *Replace taxon ids*

Description

Replace taxon ids in a [taxmap()] or [taxonomy()] object.

```
obj$replace_taxon_ids(new_ids)
replace_taxon_ids(obj, new_ids)
```

Arguments

obj	The [taxonomy()] or [taxmap()] object.
new_ids	A vector of new ids, one per taxon. They must be unique and in the same order as the corresponding ids in 'obj$taxon_ids()'.

Value

A [taxonomy()] or [taxmap()] object with new taxon ids

Examples

```
# Replace taxon IDs with numbers
replace_taxon_ids(ex_taxmap, seq_len(length(ex_taxmap$taxa)))

# Make taxon IDs capital letters
replace_taxon_ids(ex_taxmap, toupper(taxon_ids(ex_taxmap)))
```

reverse *Reverse sequences*

Description

Find the reverse of one or more sequences stored as a character vector. This is a wrapper for `rev` for character vectors instead of lists of character vectors with one value per letter.

Usage

```
reverse(seqs)
```

Arguments

seqs	A character vector with one element per sequence.

See Also

Other sequence transformations: `complement()`, `rev_comp()`

Examples

```
reverse(c("aagtgGGTGaa", "AAGTGGT"))
```

rev_comp *Revere complement sequences*

Description

Make the reverse complement of one or more sequences stored as a character vector. This is a wrapper for `comp` for character vectors instead of lists of character vectors with one value per letter. IUPAC ambiguity codes are handled and the upper/lower case is preserved.

Usage

```
rev_comp(seqs)
```

Arguments

seqs	A character vector with one element per sequence.

See Also

Other sequence transformations: `complement()`, `reverse()`

Examples

```
rev_comp(c("aagtgGGTGaa", "AAGTGGT"))
```

`roots` *Get root taxa*

Description

Return the root taxa for a [taxonomy()] or [taxmap()] object. Can also be used to get the roots of a subset of taxa.

```
obj$roots(subset = NULL, value = "taxon_indexes")
roots(obj, subset = NULL, value = "taxon_indexes")
```

Arguments

`obj`	The [taxonomy()] or [taxmap()] object containing taxon information to be queried.
`subset`	Taxon IDs, TRUE/FALSE vector, or taxon indexes to find roots for. Default: All taxa in 'obj' will be used. Any variable name that appears in [all_names()] can be used as if it was a vector on its own.
`value`	What data to return. This is usually the name of column in a table in 'obj$data'. Any result of 'all_names(obj)' can be used, but it usually only makes sense to data that corresponds to taxa 1:1, such as [taxon_ranks()]. By default, taxon indexes are returned.

Value

'character'

See Also

Other taxonomy indexing functions: `branches()`, `internodes()`, `leaves()`, `stems()`, `subtaxa()`, `supertaxa()`

Examples

```
# Return indexes of root taxa
roots(ex_taxmap)

# Return indexes for a subset of taxa
roots(ex_taxmap, subset = 2:17)

# Return something besides taxon indexes
roots(ex_taxmap, value = "taxon_names")
```

`sample_frac_obs` *Sample a proportion of observations from [taxmap()]*

Description

Randomly sample some proportion of observations from a [taxmap()] object. Weights can be specified for observations or their taxa. See [dplyr::sample_frac()] for the inspiration for this function. Calling the function using the ‘obj$sample_frac_obs(...)‘ style edits "obj" in place, unlike most R functions. However, calling the function using the ‘sample_frac_obs(obj, ...)‘ imitates R’s traditional copy-on-modify semantics, so "obj" would not be changed; instead a changed version would be returned, like most R functions.

```
obj$sample_frac_obs(data, size, replace = FALSE,
  taxon_weight = NULL, obs_weight = NULL,
  use_supertaxa = TRUE, collapse_func = mean, ...)
sample_frac_obs(obj, data, size, replace = FALSE,
  taxon_weight = NULL, obs_weight = NULL,
  use_supertaxa = TRUE, collapse_func = mean, ...)
```

Arguments

`obj`	([taxmap()]) The object to sample from.
`data`	Dataset names, indexes, or a logical vector that indicates which datasets in ‘obj$data‘ to sample. If multiple datasets are sample at once, then they must be the same length.
`size`	(‘numeric‘ of length 1) The proportion of observations to sample.
`replace`	(‘logical‘ of length 1) If ‘TRUE‘, sample with replacement.
`taxon_weight`	(‘numeric‘) Non-negative sampling weights of each taxon. If ‘use_supertaxa‘ is ‘TRUE‘, the weights for each taxon in an observation’s classification are supplied to ‘collapse_func‘ to get the observation weight. If ‘obs_weight‘ is also specified, the two weights are multiplied (after ‘taxon_weight‘ for each observation is calculated).
`obs_weight`	(‘numeric‘) Sampling weights of each observation. If ‘taxon_weight‘ is also specified, the two weights are multiplied (after ‘taxon_weight‘ for each observation is calculated).
`use_supertaxa`	(‘logical‘ or ‘numeric‘ of length 1) Affects how the ‘taxon_weight‘ is used. If ‘TRUE‘, the weights for each taxon in an observation’s classification are multiplied to get the observation weight. If ‘FALSE‘ just the taxonomic level the observation is assign to it considered. Positive numbers indicate the number of ranks above the each taxon to use. ‘0‘ is equivalent to ‘FALSE‘. Negative numbers are equivalent to ‘TRUE‘.
`collapse_func`	(‘function‘ of length 1) If ‘taxon_weight‘ option is used and ‘supertaxa‘ is ‘TRUE‘, the weights for each taxon in an observation’s classification are supplied to ‘collapse_func‘ to get the observation weight. This function should take numeric vector and return a single number.

...	Additional options are passed to [filter_obs()].
target	DEPRECIATED. use "data" instead.

Value

An object of type [taxmap()]

See Also

Other taxmap manipulation functions: arrange_obs(), arrange_taxa(), filter_obs(), filter_taxa(), mutate_obs(), sample_frac_taxa(), sample_n_obs(), sample_n_taxa(), select_obs(), transmute_obs()

Examples

```
# Sample half of the rows fram a table
sample_frac_obs(ex_taxmap, "info", 0.5)

# Sample multiple datasets at once
sample_frac_obs(ex_taxmap, c("info", "phylopic_ids", "foods"), 0.5)
```

---

sample_frac_taxa *Sample a proportion of taxa from [taxonomy()] or [taxmap()]*

Description

Randomly sample some proportion of taxa from a [taxonomy()] or [taxmap()] object. Weights can be specified for taxa or the observations assigned to them. See [dplyr::sample_frac()] for the inspiration for this function.

```
obj$sample_frac_taxa(size, taxon_weight = NULL,
  obs_weight = NULL, obs_target = NULL,
  use_subtaxa = TRUE, collapse_func = mean, ...)
sample_frac_taxa(obj, size, taxon_weight = NULL,
  obs_weight = NULL, obs_target = NULL,
  use_subtaxa = TRUE, collapse_func = mean, ...)
```

Arguments

obj	([taxonomy()] or [taxmap()]) The object to sample from.
size	(‘numeric‘ of length 1) The proportion of taxa to sample.
taxon_weight	(‘numeric‘) Non-negative sampling weights of each taxon. If ‘obs_weight‘ is also specified, the two weights are multiplied (after ‘obs_weight‘ for each taxon is calculated).

`obs_weight` (‘numeric‘) This option only applies to [taxmap()] objects. Sampling weights of each observation. The weights for each observation assigned to a given taxon are supplied to ‘collapse_func‘ to get the taxon weight. If ‘use_subtaxa‘ is ‘TRUE‘ then the observations assigned to every subtaxa are also used. Any variable name that appears in [all_names()] can be used as if it was a vector on its own. If ‘taxon_weight‘ is also specified, the two weights are multiplied (after ‘obs_weight‘ for each observation is calculated). ‘obs_target‘ must be used with this option.

`obs_target` (‘character‘ of length 1) This option only applies to [taxmap()] objects. The name of the data set in ‘obj$data‘ that values in ‘obs_weight‘ corresponds to. Must be used when ‘obs_weight‘ is used.

`use_subtaxa` (‘logical‘ or ‘numeric‘ of length 1) Affects how the ‘obs_weight‘ option is used. If ‘TRUE‘, the weights for each taxon in an observation’s classification are multiplied to get the observation weight. If ‘TRUE‘ just the taxonomic level the observation is assign to it considered. Positive numbers indicate the number of ranks below the target taxa to return. ‘0‘ is equivalent to ‘FALSE‘. Negative numbers are equivalent to ‘TRUE‘.

`collapse_func` (‘function‘ of length 1) If ‘taxon_weight‘ is used and ‘supertaxa‘ is ‘TRUE‘, the weights for each taxon in an observation’s classification are supplied to ‘collapse_func‘ to get the observation weight. This function should take numeric vector and return a single number.

`...` Additional options are passed to [filter_taxa()].

Value

An object of type [taxonomy()] or [taxmap()]

See Also

Other taxmap manipulation functions: `arrange_obs()`, `arrange_taxa()`, `filter_obs()`, `filter_taxa()`, `mutate_obs()`, `sample_frac_obs()`, `sample_n_obs()`, `sample_n_taxa()`, `select_obs()`, `transmute_obs()`

Examples

```
# sample half of the taxa
sample_frac_taxa(ex_taxmap, 0.5, supertaxa = TRUE)
```

`sample_n_obs` *Sample n observations from [taxmap()]*

Description

Randomly sample some number of observations from a [taxmap()] object. Weights can be specified for observations or the taxa they are classified by. Any variable name that appears in [all_names()] can be used as if it was a vector on its own. See [dplyr::sample_n()] for the inspiration for this function. Calling the function using the ‘obj$sample_n_obs(...)‘ style edits "obj" in place, unlike most R functions. However, calling the function using the ‘sample_n_obs(obj, ...)‘ imitates R’s traditional copy-on-modify semantics, so "obj" would not be changed; instead a changed version would be returned, like most R functions.

```
obj$sample_n_obs(data, size, replace = FALSE,
  taxon_weight = NULL, obs_weight = NULL,
  use_supertaxa = TRUE, collapse_func = mean, ...)
sample_n_obs(obj, data, size, replace = FALSE,
  taxon_weight = NULL, obs_weight = NULL,
  use_supertaxa = TRUE, collapse_func = mean, ...)
```

Arguments

`obj`	([taxmap()]) The object to sample from.
`data`	Dataset names, indexes, or a logical vector that indicates which datasets in ‘obj$data‘ to sample. If multiple datasets are sampled at once, then they must be the same length.
`size`	(‘numeric‘ of length 1) The number of observations to sample.
`replace`	(‘logical‘ of length 1) If ‘TRUE‘, sample with replacement.
`taxon_weight`	(‘numeric‘) Non-negative sampling weights of each taxon. If ‘use_supertaxa‘ is ‘TRUE‘, the weights for each taxon in an observation’s classification are supplied to ‘collapse_func‘ to get the observation weight. If ‘obs_weight‘ is also specified, the two weights are multiplied (after ‘taxon_weight‘ for each observation is calculated).
`obs_weight`	(‘numeric‘) Sampling weights of each observation. If ‘taxon_weight‘ is also specified, the two weights are multiplied (after ‘taxon_weight‘ for each observation is calculated).
`use_supertaxa`	(‘logical‘ or ‘numeric‘ of length 1) Affects how the ‘taxon_weight‘ is used. If ‘TRUE‘, the weights for each taxon in an observation’s classification are multiplied to get the observation weight. Otherwise, just the taxonomic level the observation is assign to it considered. If ‘TRUE‘, use all supertaxa. Positive numbers indicate the number of ranks above each taxon to use. ‘0‘ is equivalent to ‘FALSE‘. Negative numbers are equivalent to ‘TRUE‘.
`collapse_func`	(‘function‘ of length 1) If ‘taxon_weight‘ option is used and ‘supertaxa‘ is ‘TRUE‘, the weights for each taxon in an observation’s classification are supplied to ‘collapse_func‘ to get the observation weight. This function should take numeric vector and return a single number.
`...`	Additional options are passed to [filter_obs()].
`target`	DEPRECIATED. use "data" instead.

Value

An object of type [taxmap()]

See Also

Other taxmap manipulation functions: arrange_obs(), arrange_taxa(), filter_obs(), filter_taxa(), mutate_obs(), sample_frac_obs(), sample_frac_taxa(), sample_n_taxa(), select_obs(), transmute_obs()

Examples

```
# Sample 2 rows without replacement
sample_n_obs(ex_taxmap, "info", 2)
sample_n_obs(ex_taxmap, "foods", 2)

# Sample with replacement
sample_n_obs(ex_taxmap, "info", 10, replace = TRUE)

# Sample some rows for often then others
sample_n_obs(ex_taxmap, "info", 3, obs_weight = n_legs)

# Sample multiple datasets at once
sample_n_obs(ex_taxmap, c("info", "phylopic_ids", "foods"), 3)
```

---

sample_n_taxa *Sample n taxa from [taxonomy()] or [taxmap()]*

Description

Randomly sample some number of taxa from a [taxonomy()] or [taxmap()] object. Weights can be specified for taxa or the observations assigned to them. See [dplyr::sample_n()] for the inspiration for this function.

```
obj$sample_n_taxa(size, taxon_weight = NULL,
  obs_weight = NULL, obs_target = NULL,
  use_subtaxa = TRUE, collapse_func = mean, ...)
sample_n_taxa(obj, size, taxon_weight = NULL,
  obs_weight = NULL, obs_target = NULL,
  use_subtaxa = TRUE, collapse_func = mean, ...)
```

Arguments

obj	([taxonomy()] or [taxmap()]) The object to sample from.
size	(‘numeric‘ of length 1) The number of taxa to sample.
taxon_weight	(‘numeric‘) Non-negative sampling weights of each taxon. If ‘obs_weight‘ is also specified, the two weights are multiplied (after ‘obs_weight‘ for each taxon is calculated).

`obs_weight` (‘numeric‘) This option only applies to [taxmap()] objects. Sampling weights of each observation. The weights for each observation assigned to a given taxon are supplied to ‘collapse_func‘ to get the taxon weight. If ‘use_subtaxa‘ is ‘TRUE‘ then the observations assigned to every subtaxa are also used. Any variable name that appears in [all_names()] can be used as if it was a vector on its own. If ‘taxon_weight‘ is also specified, the two weights are multiplied (after ‘obs_weight‘ for each observation is calculated). ‘obs_target‘ must be used with this option.

`obs_target` (‘character‘ of length 1) This option only applies to [taxmap()] objects. The name of the data set in ‘obj$data‘ that values in ‘obs_weight‘ corresponds to. Must be used when ‘obs_weight‘ is used.

`use_subtaxa` (‘logical‘ or ‘numeric‘ of length 1) Affects how the ‘obs_weight‘ option is used. If ‘TRUE‘, the weights for each taxon in an observation’s classification are multiplied to get the observation weight. If ‘FALSE‘ just the taxonomic level the observation is assign to it considered. Positive numbers indicate the number of ranks below the each taxon to use. ‘0‘ is equivalent to ‘FALSE‘. Negative numbers are equivalent to ‘TRUE‘.

`collapse_func` (‘function‘ of length 1) If ‘taxon_weight‘ is used and ‘supertaxa‘ is ‘TRUE‘, the weights for each taxon in an observation’s classification are supplied to ‘collapse_func‘ to get the observation weight. This function should take numeric vector and return a single number.

`...` Additional options are passed to [filter_taxa()].

Value

An object of type [taxonomy()] or [taxmap()]

See Also

Other taxmap manipulation functions: `arrange_obs()`, `arrange_taxa()`, `filter_obs()`, `filter_taxa()`, `mutate_obs()`, `sample_frac_obs()`, `sample_frac_taxa()`, `sample_n_obs()`, `select_obs()`, `transmute_obs()`

Examples

```
# Randomly sample three taxa
sample_n_taxa(ex_taxmap, 3)

# Include supertaxa
sample_n_taxa(ex_taxmap, 3, supertaxa = TRUE)

# Include subtaxa
sample_n_taxa(ex_taxmap, 1, subtaxa = TRUE)

# Sample some taxa more often then others
sample_n_taxa(ex_taxmap, 3, supertaxa = TRUE,
              obs_weight = n_legs, obs_target = "info")
```

`select_obs` *Subset columns in a [taxmap()] object*

Description

Subsets columns in a [taxmap()] object. Takes and returns a [taxmap()] object. Any variable name that appears in [all_names()] can be used as if it was a vector on its own. See [dplyr::select()] for the inspiration for this function and more information. Calling the function using the ‘obj$select_obs(...)‘ style edits "obj" in place, unlike most R functions. However, calling the function using the ‘select_obs(obj, ...)‘ imitates R's traditional copy-on-modify semantics, so "obj" would not be changed; instead a changed version would be returned, like most R functions.

```
obj$select_obs(data, ...)
select_obs(obj, data, ...)
```

Arguments

`obj`	An object of type [taxmap()]
`data`	Dataset names, indexes, or a logical vector that indicates which tables in ‘obj$data‘ to subset columns in. Multiple tables can be subset at once.
`...`	One or more column names to return in the new object. Each can be one of two things: **expression with unquoted column name** The name of a column in the dataset typed as if it was a variable on its own. **‘numeric‘** Indexes of columns in the dataset To match column names with a character vector, use ‘matches("my_col_name")‘. To match a logical vector, convert it to a column index using ‘which‘.
`target`	DEPRECIATED. use "data" instead.

Value

An object of type [taxmap()]

See Also

Other taxmap manipulation functions: `arrange_obs()`, `arrange_taxa()`, `filter_obs()`, `filter_taxa()`, `mutate_obs()`, `sample_frac_obs()`, `sample_frac_taxa()`, `sample_n_obs()`, `sample_n_taxa()`, `transmute_obs()`

Examples

```
# Selecting a column by name
select_obs(ex_taxmap, "info", dangerous)

# Selecting a column by index
select_obs(ex_taxmap, "info", 3)
```

```
# Selecting a column by regular expressions
select_obs(ex_taxmap, "info", matches("^n"))
```

`stems` *Get stem taxa*

Description

Return the stem taxa for a [taxonomy()] or a [taxmap()] object. Stem taxa are all those from the roots to the first taxon with more than one subtaxon.

```
obj$stems(subset = NULL, simplify = FALSE,
  value = "taxon_indexes", exclude_leaves = FALSE)
stems(obj, subset = NULL, simplify = FALSE,
  value = "taxon_indexes", exclude_leaves = FALSE)
```

Arguments

`obj`	The [taxonomy()] or [taxmap()] object containing taxon information to be queried.
`subset`	Taxon IDs, TRUE/FALSE vector, or taxon indexes to find stems for. Default: All taxa in 'obj' will be used. Any variable name that appears in [all_names()] can be used as if it was a vector on its own.
`value`	What data to return. This is usually the name of column in a table in 'obj$data'. Any result of 'all_names(obj)' can be used, but it usually only makes sense to data that corresponds to taxa 1:1, such as [taxon_ranks()]. By default, taxon indexes are returned.
`simplify`	('logical') If 'TRUE', then combine all the results into a single vector of unique values.
`exclude_leaves`	('logical') If 'TRUE', the do not include taxa with no subtaxa.

Value

'character'

See Also

Other taxonomy indexing functions: `branches()`, `internodes()`, `leaves()`, `roots()`, `subtaxa()`, `supertaxa()`

Examples

```
# Return indexes of stem taxa
stems(ex_taxmap)

# Return indexes for a subset of taxa
stems(ex_taxmap, subset = 2:17)

# Return something besides taxon indexes
stems(ex_taxmap, value = "taxon_names")

# Return a vector instead of a list
stems(ex_taxmap, value = "taxon_names", simplify = TRUE)
```

---

subtaxa *Get subtaxa*

Description

Return data for the subtaxa of each taxon in an [taxonomy()] or [taxmap()] object.

```
obj$subtaxa(subset = NULL, recursive = TRUE,
  simplify = FALSE, include_input = FALSE, value = "taxon_indexes")
subtaxa(obj, subset = NULL, recursive = TRUE,
  simplify = FALSE, include_input = FALSE, value = "taxon_indexes")
```

Arguments

obj	The [taxonomy()] or [taxmap()] object containing taxon information to be queried.
subset	Taxon IDs, TRUE/FALSE vector, or taxon indexes to find subtaxa for. Default: All taxa in ‘obj‘ will be used. Any variable name that appears in [all_names()] can be used as if it was a vector on its own.
recursive	(‘logical‘ or ‘numeric‘) If ‘FALSE‘, only return the subtaxa one rank below the target taxa. If ‘TRUE‘, return all the subtaxa of every subtaxa, etc. Positive numbers indicate the number of ranks below the immediate subtaxa to return. ‘1‘ is equivalent to ‘FALSE‘. Negative numbers are equivalent to ‘TRUE‘. Since the algorithm is optimized for traversing all of large trees, ‘numeric‘ values greater than 0 for this option actually take slightly longer to compute than either TRUE or FALSE.
simplify	(‘logical‘) If ‘TRUE‘, then combine all the results into a single vector of unique values.
include_input	(‘logical‘) If ‘TRUE‘, the input taxa are included in the output
value	What data to return. This is usually the name of column in a table in ‘obj$data‘. Any result of [all_names()] can be used, but it usually only makes sense to data that corresponds to taxa 1:1, such as [taxon_ranks()]. By default, taxon indexes are returned.

Value

If 'simplify = FALSE', then a list of vectors are returned corresponding to the 'target' argument. If 'simplify = TRUE', then the unique values are returned in a single vector.

See Also

Other taxonomy indexing functions: branches(), internodes(), leaves(), roots(), stems(), supertaxa()

Examples

```
# return the indexes for subtaxa for each taxon
subtaxa(ex_taxmap)

# Only return data for some taxa using taxon indexes
subtaxa(ex_taxmap, subset = 1:3)

# Only return data for some taxa using taxon ids
subtaxa(ex_taxmap, subset = c("d", "e"))

# Only return data for some taxa using logical tests
subtaxa(ex_taxmap, subset = taxon_ranks == "genus")

# Only return subtaxa one level below
subtaxa(ex_taxmap, recursive = FALSE)

# Only return subtaxa some number of ranks below
subtaxa(ex_taxmap, recursive = 2)

# Return something besides taxon indexes
subtaxa(ex_taxmap, value = "taxon_names")
```

---

subtaxa_apply *Apply function to subtaxa of each taxon*

Description

Apply a function to the subtaxa for each taxon. This is similar to using [subtaxa()] with [lapply()] or [sapply()].

```
obj$subtaxa_apply(func, subset = NULL, recursive = TRUE,
  simplify = FALSE, include_input = FALSE, value = "taxon_indexes", ...)
subtaxa_apply(obj, func, subset = NULL, recursive = TRUE,
  simplify = FALSE, include_input = FALSE, value = "taxon_indexes", ...)
```

Arguments

obj	The [taxonomy()] or [taxmap()] object containing taxon information to be queried.
func	(‘function‘) The function to apply.
subset	Taxon IDs, TRUE/FALSE vector, or taxon indexes to use. Default: All taxa in ‘obj‘ will be used. Any variable name that appears in [all_names()] can be used as if it was a vector on its own.
recursive	(‘logical‘ or ‘numeric‘) If ‘FALSE‘, only return the subtaxa one rank below the target taxa. If ‘TRUE‘, return all the subtaxa of every subtaxa, etc. Positive numbers indicate the number of recursions (i.e. number of ranks below the target taxon to return). ‘1‘ is equivalent to ‘FALSE‘. Negative numbers are equivalent to ‘TRUE‘.
simplify	(‘logical‘) If ‘TRUE‘, then combine all the results into a single vector of unique values.
include_input	(‘logical‘) If ‘TRUE‘, the input taxa are included in the output
value	What data to give to the function. Any result of ‘all_names(obj)‘ can be used, but it usually only makes sense to use data that has an associated taxon id.
...	Extra arguments are passed to the function.

Examples

```
# Count number of subtaxa in each taxon
subtaxa_apply(ex_taxmap, length)

# Paste all the subtaxon names for each taxon
subtaxa_apply(ex_taxmap, value = "taxon_names",
              recursive = FALSE, paste0, collapse = ", ")
```

---

supertaxa *Get all supertaxa of a taxon*

Description

Return data for supertaxa (i.e. all taxa the target taxa are a part of) of each taxon in a [taxonomy()] or [taxmap()] object.

```
obj$supertaxa(subset = NULL, recursive = TRUE,
  simplify = FALSE, include_input = FALSE,
  value = "taxon_indexes", na = FALSE)
supertaxa(obj, subset = NULL, recursive = TRUE,
  simplify = FALSE, include_input = FALSE,
  value = "taxon_indexes", na = FALSE)
```

Arguments

obj	The [taxonomy()] or [taxmap()] object containing taxon information to be queried.
subset	Taxon IDs, TRUE/FALSE vector, or taxon indexes to find supertaxa for. Default: All taxa in 'obj' will be used. Any variable name that appears in [all_names()] can be used as if it was a vector on its own.
recursive	('logical' or 'numeric') If 'FALSE', only return the supertaxa one rank above the target taxa. If 'TRUE', return all the supertaxa of every supertaxa, etc. Positive numbers indicate the number of recursions (i.e. number of ranks above the target taxon to return). '1' is equivalent to 'FALSE'. Negative numbers are equivalent to 'TRUE'.
simplify	('logical') If 'TRUE', then combine all the results into a single vector of unique values.
include_input	('logical') If 'TRUE', the input taxa are included in the output
value	What data to return. Any result of [all_names()] can be used, but it usually only makes sense to use data that has an associated taxon id.
na	('logical') If 'TRUE', return 'NA' where information is not available.

Value

If 'simplify = FALSE', then a list of vectors are returned corresponding to the 'subset' argument. If 'simplify = TRUE', then unique values are returned in a single vector.

See Also

Other taxonomy indexing functions: branches(), internodes(), leaves(), roots(), stems(), subtaxa()

Examples

```
# return the indexes for supertaxa for each taxon
supertaxa(ex_taxmap)

# Only return data for some taxa using taxon indexes
supertaxa(ex_taxmap, subset = 1:3)

# Only return data for some taxa using taxon ids
supertaxa(ex_taxmap, subset = c("d", "e"))

# Only return data for some taxa using logical tests
supertaxa(ex_taxmap, subset = taxon_ranks == "species")

# Only return supertaxa one level above
supertaxa(ex_taxmap, recursive = FALSE)

# Only return supertaxa some number of ranks above
supertaxa(ex_taxmap, recursive = 2)

# Return something besides taxon indexes
supertaxa(ex_taxmap, value = "taxon_names")
```

`supertaxa_apply` *Apply function to supertaxa of each taxon*

Description

Apply a function to the supertaxa for each taxon. This is similar to using [supertaxa()] with [lapply()] or [sapply()].

```
obj$supertaxa_apply(func, subset = NULL, recursive = TRUE,
  simplify = FALSE, include_input = FALSE, value = "taxon_indexes",
  na = FALSE, ...)
supertaxa_apply(obj, func, subset = NULL, recursive = TRUE,
  simplify = FALSE, include_input = FALSE, value = "taxon_indexes",
  na = FALSE, ....)
```

Arguments

`obj`	The [taxonomy()] or [taxmap()] object containing taxon information to be queried.
`func`	(‘function‘) The function to apply.
`subset`	Taxon IDs, TRUE/FALSE vector, or taxon indexes of taxa to use. Default: All taxa in ‘obj‘ will be used. Any variable name that appears in [all_names()] can be used as if it was a vector on its own.
`recursive`	(‘logical‘ or ‘numeric‘) If ‘FALSE‘, only return the supertaxa one rank above the target taxa. If ‘TRUE‘, return all the supertaxa of every supertaxa, etc. Positive numbers indicate the number of recursions (i.e. number of ranks above the target taxon to return). ‘1‘ is equivalent to ‘FALSE‘. Negative numbers are equivalent to ‘TRUE‘.
`simplify`	(‘logical‘) If ‘TRUE‘, then combine all the results into a single vector of unique values.
`include_input`	(‘logical‘) If ‘TRUE‘, the input taxa are included in the output
`value`	What data to give to the function. Any result of ‘all_names(obj)‘ can be used, but it usually only makes sense to use data that has an associated taxon id.
`na`	(‘logical‘) If ‘TRUE‘, return ‘NA‘ where information is not available.
`...`	Extra arguments are passed to the function.

Examples

```
# Get number of supertaxa that each taxon is contained in
supertaxa_apply(ex_taxmap, length)

# Get classifications for each taxon
# Note; this can be done with `classifications()` easier
supertaxa_apply(ex_taxmap, paste, collapse = ";", include_input = TRUE,
                value = "taxon_names")
```

taxa *A class for multiple taxon objects*

Description

Stores one or more [taxon()] objects. This is just a thin wrapper for a list of [taxon()] objects.

Usage

```
taxa(..., .list = NULL)
```

Arguments

`...`	Any number of object of class [taxon()]
`.list`	An alternate to the '...' input. Any number of object of class [taxon()]. Cannot be used with '...'.

Details

This is the documentation for the class called 'taxa'. If you are looking for the documentation for the package as a whole: [taxa-package].

Value

An 'R6Class' object of class 'Taxon'

See Also

Other classes: `hierarchies()`, `hierarchy()`, `taxmap()`, `taxon()`, `taxon_database()`, `taxon_id()`, `taxon_name()`, `taxon_rank()`, `taxonomy()`

Examples

```
(a <- taxon(
  name = taxon_name("Poa annua"),
  rank = taxon_rank("species"),
  id = taxon_id(93036)
))
taxa(a, a, a)

# a null set
x <- taxon(NULL)
taxa(x, x, x)

# combo non-null and null
taxa(a, x, a)
```

taxmap *Taxmap class*

Description

A class designed to store a taxonomy and associated information. This class builds on the [taxonomy()] class. User defined data can be stored in the list 'obj$data', where 'obj' is a taxmap object. Data that is associated with taxa can be manipulated in a variety of ways using functions like [filter_taxa()] and [filter_obs()]. To associate the items of lists/vectors with taxa, name them by [taxon_ids()]. For tables, add a column named 'taxon_id' that stores [taxon_ids()].

Usage

```
taxmap(..., .list = NULL, data = NULL, funcs = list(), named_by_rank = FALSE)
```

Arguments

`...`	Any number of object of class [hierarchy()] or character vectors.
`.list`	An alternate to the '...' input. Any number of object of class [hierarchy()] or character vectors in a list. Cannot be used with '...'.
`data`	A list of tables with data associated with the taxa.
`funcs`	A named list of functions to include in the class. Referring to the names of these in functions like [filter_taxa()] will execute the function and return the results. If the function has at least one argument, the taxmap object is passed to it.
`named_by_rank`	('TRUE'/'FALSE') If 'TRUE' and the input is a list of vectors with each vector named by ranks, include that rank info in the output object, so it can be accessed by 'out$taxon_ranks()'. If 'TRUE', taxa with different ranks, but the same name and location in the taxonomy, will be considered different taxa.

Details

To initialize a 'taxmap' object with associated data sets, use the parsing functions [parse_tax_data()], [lookup_tax_data()], and [extract_tax_data()].

on initialize, function sorts the taxon list based on rank (if rank information is available), see [ranks_ref] for the reference rank names and orders

Value

An 'R6Class' object of class [taxmap()]

See Also

Other classes: `hierarchies()`, `hierarchy()`, `taxa()`, `taxon()`, `taxon_database()`, `taxon_id()`, `taxon_name()`, `taxon_rank()`, `taxonomy()`

Examples

```
# The code below shows how to contruct a taxmap object from scratch.
# Typically, taxmap objects would be the output of a parsing function,
#  not created from scratch, but this is for demostration purposes.

notoryctidae <- taxon(
name = taxon_name("Notoryctidae"),
rank = taxon_rank("family"),
id = taxon_id(4479)
)
notoryctes <- taxon(
  name = taxon_name("Notoryctes"),
  rank = taxon_rank("genus"),
  id = taxon_id(4544)
)
typhlops <- taxon(
  name = taxon_name("typhlops"),
  rank = taxon_rank("species"),
  id = taxon_id(93036)
)
mammalia <- taxon(
  name = taxon_name("Mammalia"),
  rank = taxon_rank("class"),
  id = taxon_id(9681)
)
felidae <- taxon(
  name = taxon_name("Felidae"),
  rank = taxon_rank("family"),
  id = taxon_id(9681)
)
felis <- taxon(
  name = taxon_name("Felis"),
  rank = taxon_rank("genus"),
  id = taxon_id(9682)
)
catus <- taxon(
  name = taxon_name("catus"),
  rank = taxon_rank("species"),
  id = taxon_id(9685)
)
panthera <- taxon(
  name = taxon_name("Panthera"),
  rank = taxon_rank("genus"),
  id = taxon_id(146712)
)
tigris <- taxon(
  name = taxon_name("tigris"),
  rank = taxon_rank("species"),
  id = taxon_id(9696)
)
plantae <- taxon(
  name = taxon_name("Plantae"),
```

```
  rank = taxon_rank("kingdom"),
  id = taxon_id(33090)
)
solanaceae <- taxon(
  name = taxon_name("Solanaceae"),
  rank = taxon_rank("family"),
  id = taxon_id(4070)
)
solanum <- taxon(
  name = taxon_name("Solanum"),
  rank = taxon_rank("genus"),
  id = taxon_id(4107)
)
lycopersicum <- taxon(
  name = taxon_name("lycopersicum"),
  rank = taxon_rank("species"),
  id = taxon_id(49274)
)
tuberosum <- taxon(
  name = taxon_name("tuberosum"),
  rank = taxon_rank("species"),
  id = taxon_id(4113)
)
homo <- taxon(
  name = taxon_name("homo"),
  rank = taxon_rank("genus"),
  id = taxon_id(9605)
)
sapiens <- taxon(
  name = taxon_name("sapiens"),
  rank = taxon_rank("species"),
  id = taxon_id(9606)
)
hominidae <- taxon(
  name = taxon_name("Hominidae"),
  rank = taxon_rank("family"),
  id = taxon_id(9604)
)
unidentified <- taxon(
  name = taxon_name("unidentified")
)

tiger <- hierarchy(mammalia, felidae, panthera, tigris)
cat <- hierarchy(mammalia, felidae, felis, catus)
human <- hierarchy(mammalia, hominidae, homo, sapiens)
mole <- hierarchy(mammalia, notoryctidae, notoryctes, typhlops)
tomato <- hierarchy(plantae, solanaceae, solanum, lycopersicum)
potato <- hierarchy(plantae, solanaceae, solanum, tuberosum)
potato_partial <- hierarchy(solanaceae, solanum, tuberosum)
unidentified_animal <- hierarchy(mammalia, unidentified)
unidentified_plant <- hierarchy(plantae, unidentified)

info <- data.frame(stringsAsFactors = FALSE,
```

```
                    name = c("tiger", "cat", "mole", "human", "tomato", "potato"),
                    n_legs = c(4, 4, 4, 2, 0, 0),
                    dangerous = c(TRUE, FALSE, FALSE, TRUE, FALSE, FALSE))

abund <- data.frame(code = rep(c("T", "C", "M", "H"), 2),
                    sample_id = rep(c("A", "B"), each = 2),
                    count = c(1,2,5,2,6,2,4,0),
                    taxon_index = rep(1:4, 2))

phylopic_ids <- c("e148eabb-f138-43c6-b1e4-5cda2180485a",
                  "12899ba0-9923-4feb-a7f9-758c3c7d5e13",
                  "11b783d5-af1c-4f4e-8ab5-a51470652b47",
                  "9fae30cd-fb59-4a81-a39c-e1826a35f612",
                  "b6400f39-345a-4711-ab4f-92fd4e22cb1a",
                  "63604565-0406-460b-8cb8-1abe954b3f3a")

foods <- list(c("mammals", "birds"),
              c("cat food", "mice"),
              c("insects"),
              c("Most things, but especially anything rare or expensive"),
              c("light", "dirt"),
              c("light", "dirt"))

reaction <- function(x) {
  ifelse(x$data$info$dangerous,
         paste0("Watch out! That ", x$data$info$name, " might attack!"),
         paste0("No worries; its just a ", x$data$info$name, "."))
}

ex_taxmap <- taxmap(tiger, cat, mole, human, tomato, potato,
                    data = list(info = info,
                                phylopic_ids = phylopic_ids,
                                foods = foods,
                                abund = abund),
                    funcs = list(reaction = reaction))
```

---

taxon *Taxon class*

Description

A class used to define a single taxon. Most other classes in the taxa package include one or more objects of this class.

Usage

```
taxon(name, rank = NULL, id = NULL, authority = NULL)
```

Arguments

name	a TaxonName object [taxon_name()] or character string. if character passed in, we'll coerce to a TaxonName object internally, required
rank	a TaxonRank object [taxon_rank()] or character string. if character passed in, we'll coerce to a TaxonRank object internally, required
id	a TaxonId object [taxon_id()], numeric/integer, or character string. if numeric/integer/character passed in, we'll coerce to a TaxonId object internally, required
authority	(character) a character string, optional

Details

Note that there is a special use case of this function - you can pass 'NULL' as the first parameter to get an empty 'taxon' object. It makes sense to retain the original behavior where nothing passed in to the first parameter leads to an error, and thus creating a 'NULL' taxon is done very explicitly.

Value

An 'R6Class' object of class 'Taxon'

See Also

Other classes: hierarchies(), hierarchy(), taxa(), taxmap(), taxon_database(), taxon_id(), taxon_name(), taxon_rank(), taxonomy()

Examples

```
(x <- taxon(
  name = taxon_name("Poa annua"),
  rank = taxon_rank("species"),
  id = taxon_id(93036)
))
x$name
x$rank
x$id

# a null taxon object
taxon(NULL)
## with all NULL objects from the other classes
taxon(
  name = taxon_name(NULL),
  rank = taxon_rank(NULL),
  id = taxon_id(NULL)
)
```

taxonomy *Taxonomy class*

Description

Stores a taxonomy composed of [taxon()] objects organized in a tree structure. This differs from the [hierarchies()] class in how the [taxon()] objects are stored. Unlike [hierarchies()], each taxon is only stored once and the relationships between taxa are stored in an [edge list](https://en.wikipedia.org/wiki/Adjacency_list).

Usage

```
taxonomy(..., .list = NULL, named_by_rank = FALSE)
```

Arguments

`...`	Any number of object of class [hierarchy()] or character vectors.
`.list`	An alternate to the '...' input. Any number of object of class [hierarchy()] or character vectors in a list. Cannot be used with '...'.
`named_by_rank`	('TRUE'/'FALSE') If 'TRUE' and the input is a list of vectors with each vector named by ranks, include that rank info in the output object, so it can be accessed by 'out$taxon_ranks()'. If 'TRUE', taxa with different ranks, but the same name and location in the taxonomy, will be considered different taxa.

Value

An 'R6Class' object of class 'Taxonomy'

See Also

Other classes: `hierarchies()`, `hierarchy()`, `taxa()`, `taxmap()`, `taxon()`, `taxon_database()`, `taxon_id()`, `taxon_name()`, `taxon_rank()`

Examples

```
# Making a taxonomy object with vectors
taxonomy(c("mammalia", "felidae", "panthera", "tigris"),
         c("mammalia", "felidae", "panthera", "leo"),
         c("mammalia", "felidae", "felis", "catus"))

# Making a taxonomy object from scratch
#   Note: This information would usually come from a parsing function.
#         This is just for demonstration.
x <- taxon(
  name = taxon_name("Notoryctidae"),
  rank = taxon_rank("family"),
  id = taxon_id(4479)
)
y <- taxon(
```

```
  name = taxon_name("Notoryctes"),
  rank = taxon_rank("genus"),
  id = taxon_id(4544)
)
z <- taxon(
  name = taxon_name("Notoryctes typhlops"),
  rank = taxon_rank("species"),
  id = taxon_id(93036)
)

a <- taxon(
  name = taxon_name("Mammalia"),
  rank = taxon_rank("class"),
  id = taxon_id(9681)
)
b <- taxon(
  name = taxon_name("Felidae"),
  rank = taxon_rank("family"),
  id = taxon_id(9681)
)

cc <- taxon(
  name = taxon_name("Puma"),
  rank = taxon_rank("genus"),
  id = taxon_id(146712)
)
d <- taxon(
  name = taxon_name("Puma concolor"),
  rank = taxon_rank("species"),
  id = taxon_id(9696)
)

m <- taxon(
  name = taxon_name("Panthera"),
  rank = taxon_rank("genus"),
  id = taxon_id(146712)
)
n <- taxon(
  name = taxon_name("Panthera tigris"),
  rank = taxon_rank("species"),
  id = taxon_id(9696)
)

(hier1 <- hierarchy(z, y, x, a))
(hier2 <- hierarchy(cc, b, a, d))
(hier3 <- hierarchy(n, m, b, a))

(hrs <- hierarchies(hier1, hier2, hier3))

ex_taxonomy <- taxonomy(hier1, hier2, hier3)
```

taxonomy_table *Convert taxonomy info to a table*

Description

Convert per-taxon information, like taxon names, to a table of taxa (rows) by ranks (columns).

Arguments

obj	A taxonomy or taxmap object
subset	Taxon IDs, TRUE/FALSE vector, or taxon indexes to find supertaxa for. Default: All leaves will be used. Any variable name that appears in [all_names()] can be used as if it was a vector on its own.
value	What data to return. Default is taxon names. Any result of [all_names()] can be used, but it usually only makes sense to use data with one value per taxon, like taxon names.
use_ranks	Which ranks to use. Must be one of the following: * 'NULL' (the default): If there is rank information, use the ranks that appear in the lineage with the most ranks. Otherwise, assume the number of supertaxa corresponds to rank and use placeholders for the rank column names in the output. * 'TRUE': Use the ranks that appear in the lineage with the most ranks. An error will occur if no rank information is available. * 'FALSE': Assume the number of supertaxa corresponds to rank and use placeholders for the rank column names in the output. Do not use included rank information. * 'character': The names of the ranks to use. Requires included rank information. * 'numeric': The "depth" of the ranks to use. These are equal to 'n_supertaxa' + 1.
add_id_col	If 'TRUE', include a taxon ID column.

Value

A tibble of taxa (rows) by ranks (columns).

Examples

```
# Make a table of taxon names
taxonomy_table(ex_taxmap)

# Use a differnt value
taxonomy_table(ex_taxmap, value = "taxon_ids")

# Return a subset of taxa
taxonomy_table(ex_taxmap, subset = taxon_ranks == "genus")

# Use arbitrary ranks names based on depth
taxonomy_table(ex_taxmap, use_ranks = FALSE)
```

taxon_database *Taxonomy database class*

Description

Used to store information about taxonomy databases. This is typically used to store where taxon information came from in [taxon()] objects.

Usage

```
taxon_database(name = NULL, url = NULL, description = NULL, id_regex = NULL)
```

Arguments

name	(character) name of the database
url	(character) url for the database
description	(character) description of the database
id_regex	(character) id regex

Value

An 'R6Class' object of class 'TaxonDatabase'

See Also

[database_list]

Other classes: hierarchies(), hierarchy(), taxa(), taxmap(), taxon(), taxon_id(), taxon_name(), taxon_rank(), taxonomy()

Examples

```
# create a database entry
(x <- taxon_database(
  "ncbi",
  "http://www.ncbi.nlm.nih.gov/taxonomy",
  "NCBI Taxonomy Database",
  "*"
))
x$name
x$url

# use pre-created database objects
database_list
database_list$ncbi
```

taxon_id *Taxon ID class*

Description

Used to store taxon IDs, either arbitrary or from a taxonomy database. This is typically used to store taxon IDs in [taxon()] objects.

Usage

```
taxon_id(id, database = NULL)
```

Arguments

id	(character/integer/numeric) a taxonomic id, required
database	(database) database class object, optional

Value

An 'R6Class' object of class 'TaxonId'

See Also

Other classes: hierarchies(), hierarchy(), taxa(), taxmap(), taxon(), taxon_database(), taxon_name(), taxon_rank(), taxonomy()

Examples

```
(x <- taxon_id(12345))
x$id
x$database

(x <- taxon_id(
  12345,
  database_list$ncbi
))
x$id
x$database

# a null taxon_name object
taxon_name(NULL)
```

taxon_ids *Get taxon IDs*

Description

Return the taxon IDs in a [taxonomy()] or [taxmap()] object. They are in the order they appear in the edge list.

```
obj$taxon_ids()
taxon_ids(obj)
```

Arguments

obj	The [taxonomy()] or [taxmap()] object.

See Also

Other taxonomy data functions: classifications(), id_classifications(), is_branch(), is_internode(), is_leaf(), is_root(), is_stem(), map_data(), map_data_(), n_leaves(), n_leaves_1(), n_subtaxa(), n_subtaxa_1(), n_supertaxa(), n_supertaxa_1(), taxon_indexes(), taxon_names(), taxon_ranks()

Examples

```
# Return the taxon IDs for each taxon
taxon_ids(ex_taxmap)

# Filter using taxon IDs
filter_taxa(ex_taxmap, ! taxon_ids %in% c("c", "d"))
```

taxon_indexes *Get taxon indexes*

Description

Return the taxon indexes in a [taxonomy()] or [taxmap()] object. They are the indexes of the edge list rows.

```
obj$taxon_indexes()
taxon_indexes(obj)
```

Arguments

obj	The [taxonomy()] or [taxmap()] object.

See Also

Other taxonomy data functions: classifications(), id_classifications(), is_branch(), is_internode(), is_leaf(), is_root(), is_stem(), map_data(), map_data_(), n_leaves(), n_leaves_1(), n_subtaxa(), n_subtaxa_1(), n_supertaxa(), n_supertaxa_1(), taxon_ids(), taxon_names(), taxon_ranks()

Examples

```
# Return the indexes for each taxon
taxon_indexes(ex_taxmap)

# Use in another function (stupid example; 1:5 would work too)
filter_taxa(ex_taxmap, taxon_indexes < 5)
```

---

taxon_name *Taxon name class*

Description

Used to store the name of taxa. This is typically used to store where taxon names in [taxon()] objects.

Usage

```
taxon_name(name, database = NULL)
```

Arguments

name	(character) a taxonomic name. required
database	(character) database class object, optional

Value

An 'R6Class' object of class 'TaxonName'

See Also

Other classes: hierarchies(), hierarchy(), taxa(), taxmap(), taxon(), taxon_database(), taxon_id(), taxon_rank(), taxonomy()

Examples

```
(poa <- taxon_name("Poa"))
(undef <- taxon_name("undefined"))
(sp1 <- taxon_name("species 1"))
(poa_annua <- taxon_name("Poa annua"))
(x <- taxon_name("Poa annua L."))

x$name
x$database

(x <- taxon_name(
  "Poa annua",
  database_list$ncbi
))
x$rank
x$database

# a null taxon_name object
taxon_name(NULL)
```

---

taxon_names *Get taxon names*

Description

Return the taxon names in a [taxonomy()] or [taxmap()] object. They are in the order they appear in the edge list.

```
obj$taxon_names()
taxon_names(obj)
```

Arguments

obj	The [taxonomy()] or [taxmap()] object.

See Also

Other taxonomy data functions: classifications(), id_classifications(), is_branch(), is_internode(), is_leaf(), is_root(), is_stem(), map_data(), map_data_(), n_leaves(), n_leaves_1(), n_subtaxa(), n_subtaxa_1(), n_supertaxa(), n_supertaxa_1(), taxon_ids(), taxon_indexes(), taxon_ranks()

Examples

```
# Return the names for each taxon
taxon_names(ex_taxmap)

# Filter by taxon name
filter_taxa(ex_taxmap, taxon_names == "Felidae", subtaxa = TRUE)
```

taxon_rank *Taxon rank class*

Description

Stores the rank of a taxon. This is typically used to store where taxon information came from in [taxon()] objects.

Usage

```
taxon_rank(name, database = NULL)
```

Arguments

name	(character) rank name. required
database	(character) database class object, optional

Value

An 'R6Class' object of class 'TaxonRank'

See Also

Other classes: hierarchies(), hierarchy(), taxa(), taxmap(), taxon(), taxon_database(), taxon_id(), taxon_name(), taxonomy()

Examples

```
taxon_rank("species")
taxon_rank("genus")
taxon_rank("kingdom")

(x <- taxon_rank(
  "species",
  database_list$ncbi
))
x$rank
x$database

# a null taxon_name object
taxon_name(NULL)
```

taxon_ranks *Get taxon ranks*

Description

Return the taxon ranks in a [taxonomy()] or [taxmap()] object. They are in the order taxa appear in the edge list.

```
obj$taxon_ranks()
taxon_ranks(obj)
```

Arguments

obj	The [taxonomy()] or [taxmap()] object.

See Also

Other taxonomy data functions: classifications(), id_classifications(), is_branch(), is_internode(), is_leaf(), is_root(), is_stem(), map_data(), map_data_(), n_leaves(), n_leaves_1(), n_subtaxa(), n_subtaxa_1(), n_supertaxa(), n_supertaxa_1(), taxon_ids(), taxon_indexes(), taxon_names()

Examples

```
# Get ranks for each taxon
taxon_ranks(ex_taxmap)

# Filter by rank
filter_taxa(ex_taxmap, taxon_ranks == "family", supertaxa = TRUE)
```

transmute_obs *Replace columns in [taxmap()] objects*

Description

Replace columns of tables in ‘obj$data‘ in [taxmap()] objects. See [dplyr::transmute()] for the inspiration for this function and more information. Calling the function using the ‘obj$transmute_obs(...)‘ style edits "obj" in place, unlike most R functions. However, calling the function using the ‘transmute_obs(obj, ...)‘ imitates R’s traditional copy-on-modify semantics, so "obj" would not be changed; instead a changed version would be returned, like most R functions.

```
obj$transmute_obs(data, ...)
transmute_obs(obj, data, ...)
```

Arguments

obj	An object of type [taxmap()]
data	Dataset name, index, or a logical vector that indicates which dataset in 'obj$data' to use.
...	One or more named columns to add. Newly created columns can be referenced in the same function call. Any variable name that appears in [all_names()] can be used as if it was a vector on its own.
target	DEPRECIATED. use "data" instead.

Value

An object of type [taxmap()]

See Also

Other taxmap manipulation functions: arrange_obs(), arrange_taxa(), filter_obs(), filter_taxa(), mutate_obs(), sample_frac_obs(), sample_frac_taxa(), sample_n_obs(), sample_n_taxa(), select_obs()

Examples

```
# Replace columns in a table with new columns
transmute_obs(ex_taxmap, "info", new_col = paste0(name, "!!!"))
```

---

write_greengenes *Write an imitation of the Greengenes database*

Description

Attempts to save taxonomic and sequence information of a taxmap object in the Greengenes output format. If the taxmap object was created using parse_greengenes, then it should be able to replicate the format exactly with the default settings.

Usage

```
write_greengenes(
  obj,
  tax_file = NULL,
  seq_file = NULL,
  tax_names = obj$get_data("taxon_names")[[1]],
  ranks = obj$get_data("gg_rank")[[1]],
  ids = obj$get_data("gg_id")[[1]],
  sequences = obj$get_data("gg_seq")[[1]]
)
```

Arguments

obj	A taxmap object
tax_file	(character of length 1) The file path to save the taxonomy file.
seq_file	(character of length 1) The file path to save the sequence fasta file. This is optional.
tax_names	(character named by taxon ids) The names of taxa
ranks	(character named by taxon ids) The ranks of taxa
ids	(character named by taxon ids) Sequence ids
sequences	(character named by taxon ids) Sequences

Details

The taxonomy output file has a format like:

```
228054  k__Bacteria; p__Cyanobacteria; c__Synechococcophycideae; o__Synech...
844608  k__Bacteria; p__Cyanobacteria; c__Synechococcophycideae; o__Synech...
...
```

The optional sequence file has a format like:

```
>1111886
AACGAACGCTGGCGGCATGCCTAACACATGCAAGTCGAACGAGACCTTCGGGTCTAGTGGCGCACGGGTGCGTA...
>1111885
AGAGTTTGATCCTGGCTCAGAATGAACGCTGGCGGCGTGCCTAACACATGCAAGTCGTACGAGAAATCCCGAGC...
...
```

See Also

Other writers: make_dada2_asv_table(), make_dada2_tax_table(), write_mothur_taxonomy(), write_rdp(), write_silva_fasta(), write_unite_general()

---

write_mothur_taxonomy *Write an imitation of the Mothur taxonomy file*

Description

Attempts to save taxonomic information of a taxmap object in the mothur '*.taxonomy' format. If the taxmap object was created using parse_mothur_taxonomy, then it should be able to replicate the format exactly with the default settings.

Usage

```
write_mothur_taxonomy(
  obj,
  file,
  tax_names = obj$get_data("taxon_names")[[1]],
  ids = obj$get_data("sequence_id")[[1]],
  scores = NULL
)
```

Arguments

obj	A taxmap object
file	(character of length 1) The file path to save the sequence fasta file. This is optional.
tax_names	(character named by taxon ids) The names of taxa
ids	(character named by taxon ids) Sequence ids
scores	(numeric named by taxon ids)

Details

The output file has a format like:

```
AY457915 Bacteria(100);Firmicutes(99);Clostridiales(99);Johnsone...
AY457914 Bacteria(100);Firmicutes(100);Clostridiales(100);Johnso...
AY457913 Bacteria(100);Firmicutes(100);Clostridiales(100);Johnso...
AY457912 Bacteria(100);Firmicutes(99);Clostridiales(99);Johnsone...
AY457911 Bacteria(100);Firmicutes(99);Clostridiales(98);Ruminoco...
```

or...

```
AY457915 Bacteria;Firmicutes;Clostridiales;Johnsonella_et_rel.;J...
AY457914 Bacteria;Firmicutes;Clostridiales;Johnsonella_et_rel.;J...
AY457913 Bacteria;Firmicutes;Clostridiales;Johnsonella_et_rel.;J...
AY457912 Bacteria;Firmicutes;Clostridiales;Johnsonella_et_rel.;J...
AY457911 Bacteria;Firmicutes;Clostridiales;Ruminococcus_et_rel.;...
```

See Also

Other writers: make_dada2_asv_table(), make_dada2_tax_table(), write_greengenes(), write_rdp(), write_silva_fasta(), write_unite_general()

write_rdp *Write an imitation of the RDP FASTA database*

Description

Attempts to save taxonomic and sequence information of a taxmap object in the RDP FASTA format. If the taxmap object was created using parse_rdp, then it should be able to replicate the format exactly with the default settings.

Usage

```
write_rdp(
  obj,
  file,
  tax_names = obj$get_data("taxon_names")[[1]],
  ranks = obj$get_data("rdp_rank")[[1]],
  ids = obj$get_data("rdp_id")[[1]],
  info = obj$get_data("seq_name")[[1]],
  sequences = obj$get_data("rdp_seq")[[1]]
)
```

Arguments

obj	A taxmap object
file	(character of length 1) The file path to save the sequence fasta file. This is optional.
tax_names	(character named by taxon ids) The names of taxa
ranks	(character named by taxon ids) The ranks of taxa
ids	(character named by taxon ids) Sequence ids
info	(character named by taxon ids) Info associated with sequences. In the example output shown here, this field corresponds to "Sparassis crispa; MBUH-PIRJO&ILKKA94-1587/ss5"
sequences	(character named by taxon ids) Sequences

Details

The output file has a format like:

```
>S000448483 Sparassis crispa; MBUH-PIRJO&ILKKA94-1587/ss5 Lineage=Root;rootrank;Fun...
ggattcccctagtaactgcgagtgaagcgggaagagctcaaatttaaaatctggcggcgtcctcgtcgtccgagttgtaa
tctggagaagcgacatccgcgctggaccgtgtacaagtctcttggaaaagagcgtcgtagagggtgacaatcccgtcttt
...
```

See Also

Other writers: make_dada2_asv_table(), make_dada2_tax_table(), write_greengenes(), write_mothur_taxonomy(), write_silva_fasta(), write_unite_general()

`write_silva_fasta` *Write an imitation of the SILVA FASTA database*

Description

Attempts to save taxonomic and sequence information of a taxmap object in the SILVA FASTA format. If the taxmap object was created using `parse_silva_fasta`, then it should be able to replicate the format exactly with the default settings.

Usage

```
write_silva_fasta(
  obj,
  file,
  tax_names = obj$get_data("taxon_names")[[1]],
  other_names = obj$get_data("other_name")[[1]],
  ids = obj$get_data("ncbi_id")[[1]],
  start = obj$get_data("start_pos")[[1]],
  end = obj$get_data("end_pos")[[1]],
  sequences = obj$get_data("silva_seq")[[1]]
)
```

Arguments

`obj`	A taxmap object
`file`	(`character` of length 1) The file path to save the sequence fasta file. This is optional.
`tax_names`	(`character` named by taxon ids) The names of taxa
`other_names`	(`character` named by taxon ids) Alternate names of taxa. Will be added after the primary name.
`ids`	(`character` named by taxon ids) Sequence ids
`start`	(`character`) The start position of the sequence.
`end`	(`character`) The end position of the sequence.
`sequences`	(`character` named by taxon ids) Sequences

Details

The output file has a format like:

```
>GCVF01000431.1.2369 Bacteria;Proteobacteria;Gammaproteobacteria;Oceanospiril...
CGUGCACGGUGGAUGCCUUGGCAGCCAGAGGCGAUGAAGGACGUUGUAGCCUGCGAUAAGCUCCGGUUAGGUGGCAAACA
ACCGUUUGACCCGGAGAUCUCCGAAUGGGGCAACCCACCCGUUGUAAGGCGGGUAUCACCGACUGAAUCCAUAGGUCGGU
...
```

See Also

Other writers: make_dada2_asv_table(), make_dada2_tax_table(), write_greengenes(), write_mothur_taxonomy(), write_rdp(), write_unite_general()

---

write_unite_general *Write an imitation of the UNITE general FASTA database*

Description

Attempts to save taxonomic and sequence information of a taxmap object in the UNITE general FASTA format. If the taxmap object was created using parse_unite_general, then it should be able to replicate the format exactly with the default settings.

Usage

```
write_unite_general(
  obj,
  file,
  tax_names = obj$get_data("taxon_names")[[1]],
  ranks = obj$get_data("unite_rank")[[1]],
  sequences = obj$get_data("unite_seq")[[1]],
  seq_name = obj$get_data("organism")[[1]],
  ids = obj$get_data("unite_id")[[1]],
  gb_acc = obj$get_data("acc_num")[[1]],
  type = obj$get_data("unite_type")[[1]]
)
```

Arguments

obj	A taxmap object
file	(character of length 1) The file path to save the sequence fasta file. This is optional.
tax_names	(character named by taxon ids) The names of taxa
ranks	(character named by taxon ids) The ranks of taxa
sequences	(character named by taxon ids) Sequences
seq_name	(character named by taxon ids) Name of sequences. Usually a taxon name.
ids	(character named by taxon ids) UNITE sequence ids
gb_acc	(character named by taxon ids) Genbank accession numbers
type	(character named by taxon ids) What type of sequence it is. Usually "rep" or "ref".

Details

The output file has a format like:

```
>Glomeromycota_sp|KJ484724|SH523877.07FU|reps|k__Fungi;p__Glomeromycota;c__unid...
ATAATTTGCCGAACCTAGCGTTAGCGCGAGGTTCTGCGATCAACACTTATATTTAAAACCCAACTCTTAAATTTTGTAT...
...
```

See Also

Other writers: `make_dada2_asv_table()`, `make_dada2_tax_table()`, `write_greengenes()`, `write_mothur_taxonomy()`, `write_rdp()`, `write_silva_fasta()`

---

zero_low_counts *Replace low counts with zero*

Description

For a given table in a `taxmap` object, convert all counts below a minimum number to zero. This is useful for effectively removing "singletons", "doubletons", or other low abundance counts.

Usage

```
zero_low_counts(
  obj,
  data,
  min_count = 2,
  use_total = FALSE,
  cols = NULL,
  other_cols = FALSE,
  out_names = NULL,
  dataset = NULL
)
```

Arguments

`obj`	A `taxmap` object
`data`	The name of a table in `obj$data`.
`min_count`	The minimum number of counts needed for a count to remain unchanged. Any could less than this will be converted to a zero. For example, `min_count = 2` would remove singletons.
`use_total`	If `TRUE`, the `min_count` applies to the total count for each row (e.g. OTU counts for all samples), rather than each cell in the table. For example `use_total = TRUE`, `min_count = 10` would convert all counts of any row to zero if the total for all counts in that row was less than 10.
`cols`	The columns in `data` to use. By default, all numeric columns are used. Takes one of the following inputs:

TRUE/FALSE: All/No columns will used.

Character vector: The names of columns to use

Numeric vector: The indexes of columns to use

Vector of TRUE/FALSE of length equal to the number of columns: Use the columns corresponding to `TRUE` values.

`other_cols` Preserve in the output non-target columns present in the input data. New columns will always be on the end. The "taxon_id" column will be preserved in the front. Takes one of the following inputs:

NULL: No columns will be added back, not even the taxon id column.

TRUE/FALSE: All/None of the non-target columns will be preserved.

Character vector: The names of columns to preserve

Numeric vector: The indexes of columns to preserve

Vector of TRUE/FALSE of length equal to the number of columns: Preserve the columns corresponding to `TRUE` values.

`out_names` The names of count columns in the output. Must be the same length and order as `cols` (or `unique(groups)`, if `groups` is used).

`dataset` DEPRECIATED. use "data" instead.

Value

A tibble

See Also

Other calculations: `calc_diff_abund_deseq2()`, `calc_group_mean()`, `calc_group_median()`, `calc_group_rsd()`, `calc_group_stat()`, `calc_n_samples()`, `calc_obs_props()`, `calc_prop_samples()`, `calc_taxon_abund()`, `compare_groups()`, `counts_to_presence()`, `rarefy_obs()`

Examples

```
# Parse data for examples
x = parse_tax_data(hmp_otus, class_cols = "lineage", class_sep = ";",
                   class_key = c(tax_rank = "taxon_rank", tax_name = "taxon_name"),
                   class_regex = "^(.+)__(.+)$")

# Default use
zero_low_counts(x, "tax_data")

# Use only a subset of columns
zero_low_counts(x, "tax_data", cols = c("700035949", "700097855", "700100489"))
zero_low_counts(x, "tax_data", cols = 4:6)
zero_low_counts(x, "tax_data", cols = startsWith(colnames(x$data$tax_data), "70001"))

# Including all other columns in ouput
zero_low_counts(x, "tax_data", other_cols = TRUE)

# Inlcuding specific columns in output
zero_low_counts(x, "tax_data", cols = c("700035949", "700097855", "700100489"),
```

```
                other_cols = 2:3)

# Rename output columns
zero_low_counts(x, "tax_data", cols = c("700035949", "700097855", "700100489"),
                out_names = c("a", "b", "c"))
```

Index