Package 'metacoder'

April 29, 2020

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

Version 0.3.4

Maintainer Zachary Foster <zacharyfoster1989@gmail.com>

Description A set of tools for parsing, manipulating, and graphing data classified by a hierarchy (e.g. a taxonomy).

Depends R (>= 3.0.2), taxa

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, scales, grid, taxize, seqinr, reshape2, zoo, traits, RColorBrewer, RCurl, ape, reshape, stats, grDevices, utils, lazyeval, dplyr, magrittr, readr, rlang, biomformat, phylotate, ggfittext, vegan, ggrepel, cowplot, GA, Rcpp, crayon, svglite, viridisLite, tibble

Suggests knitr, rmarkdown, testthat, zlibbioc, BiocManager, phyloseq

VignetteBuilder knitr

RoxygenNote 7.1.0

Date 2020-04-28

Encoding UTF-8

biocViews

LinkingTo Rcpp

NeedsCompilation yes

Author Zachary Foster [aut, cre], Niklaus Grunwald [ths], Rob Gilmore [ctb]

Repository CRAN

Date/Publication 2020-04-29 19:40:03 UTC

R topics documented:

`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.

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

Examples

```
## Not run:
# Install phyloseq to get example data
# source('http://bioconductor.org/biocLite.R')
# biocLite('phyloseq')

# Parse example dataset
```

```
library(phyloseq)
data(GlobalPatterns)
x <- parse_phyloseq(GlobalPatterns)

# Convert back to a phylseq object
as_phyloseq(x)


## End(Not run)
```

`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_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

```
## Not run:
# 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"))

## End(Not run)
```

`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_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

```
## Not run:
# 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"))


## End(Not run)
```

`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_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

```
## Not run:
# 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"))


## End(Not run)
```

`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_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

```
## Not run:
# 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"))


## End(Not run)
```

`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_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

```
## Not run:
# 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")

## End(Not run)
```

`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_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

```
## Not run:
# 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"))


## End(Not run)
```

`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_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

```
## Not run:
# 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")

## End(Not run)
```

`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_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

```
## Not run:
# 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')))


## End(Not run)
```

`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_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

```
## Not run:
# 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)


## End(Not run)
```

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()
```

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)
```

---

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_color_axis_label = NULL,
  node_size_axis_label = NULL,
  edge_color_axis_label = NULL,
  edge_size_axis_label = NULL,
  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_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`.

`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 `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 `nicely`. Let `igraph` choose the layout.

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

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

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

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

"mds" Use `with_mds`. Multidimensional scaling.

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

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

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

"drl" Use `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

```
## Not run:
# 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))


## End(Not run)
```

`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

```
## Not run:
# 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")


## End(Not run)
```

`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

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"))
```

---

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))
```

---

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()

`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 http://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

Author(s)

Zachary Foster and Niklaus Grunwald

`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

```
## Not run:

# 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
filter_taxa(obj, taxon_names == "Fungi", subtaxa = TRUE) %>%
  heat_tree(node_label = taxon_names, node_color = n_obs, node_size = n_obs)

## End(Not run)
```

---

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: parse_edge_list(), parse_greengenes(), parse_mothur_tax_summary(), parse_mothur_taxonomy(), parse_newick(), parse_phyloseq(), parse_phylo(), parse_qiime_biom(), parse_rdp(), parse_silva_fasta(), 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: parse_dada2(), parse_edge_list(), parse_mothur_tax_summary(), parse_mothur_taxonomy(), parse_newick(), parse_phyloseq(), parse_phylo(), parse_qiime_biom(), parse_rdp(), parse_silva_fasta(), 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: parse_dada2(), parse_edge_list(), parse_greengenes(), parse_mothur_tax_summary(), parse_newick(), parse_phyloseq(), parse_phylo(), parse_qiime_biom(), parse_rdp(), parse_silva_fasta(), 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: parse_dada2(), parse_edge_list(), parse_greengenes(), parse_mothur_taxonomy(), parse_newick(), parse_phyloseq(), parse_phylo(), parse_qiime_biom(), parse_rdp(), parse_silva_fasta(), 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: parse_dada2(), parse_edge_list(), parse_greengenes(), parse_mothur_tax_summary(), parse_mothur_taxonomy(), parse_phyloseq(), parse_phylo(), parse_qiime_biom(), parse_rdp(), parse_silva_fasta(), 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: 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_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: 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_ubiome(), parse_unite_general()

Examples

```
## Not run:

# Install phyloseq to get example data
# source('http://bioconductor.org/biocLite.R')
# biocLite('phyloseq')

# 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)


## End(Not run)
```

---

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

---

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: parse_dada2(), parse_edge_list(), 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()

---

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: parse_dada2(), parse_edge_list(), parse_greengenes(), parse_mothur_tax_summary(), parse_mothur_taxonomy(), parse_newick(), parse_phyloseq(), parse_phylo(), parse_qiime_biom(), parse_rdp(), parse_silva_fasta(), 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 (http://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:

```
         (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
```

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

```
## Not run:
# 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 ampilify
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")

## End(Not run)
```

`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 (http://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

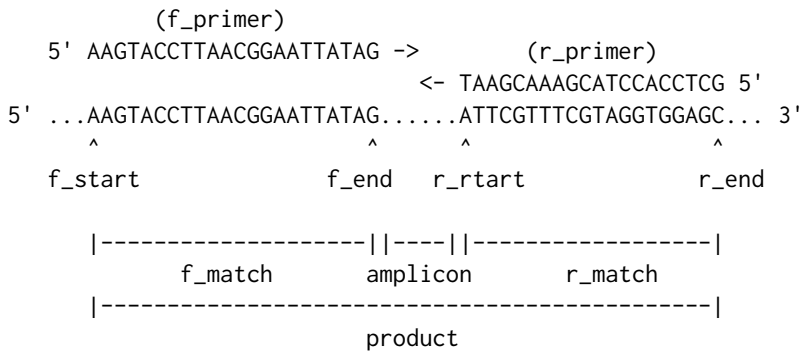
It can be confusing how the primer sequence relates to the binding sites on a reference database sequence. A simplified diagram can help. For example, if the top strand below (5' -> 3') is the database sequence, the forward primer has the same sequence as the target region, since it will bind to the 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_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

```
## Not run:

### 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$obs_apply("pcr",
                                    function(x) length(unique(x)),
                                    value = "input"))
prop_amped <- n_amplified / obj$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")


## End(Not run)
```

---

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()
```

`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_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

```
## Not run:
# 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"))


## End(Not run)
```

---

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)
```

---

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"))
```

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_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

```
## Not run:
# 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"))


## End(Not run)
```

Index