Package: glyexp (via r-universe)

May 14, 2026

Title Glycoproteomics and Glycomics Experiments

Version 0.14.1

Description Provides a tidy data framework for managing glycoproteomics and glycomics experimental data. The core feature is the 'experiment()' class, which serves as a unified data container integrating expression matrices, variable information (proteins, peptides, glycan compositions, etc.), and sample metadata (groups, batches, clinical variables, etc.). The package enforces data consistency, validates column types according to experiment types (glycomics, glycoproteomics, traitomics, traitproteomics), and provides dplyr-style data manipulation functions (filter, mutate, select, arrange, slice, join) for seamless data wrangling. As the data core of the 'glycoverse' ecosystem, it provides a consistent interface that other packages can reliably extract information from, enabling smooth data exchange and analysis workflows.

License MIT + file LICENSE

Suggests conflicted, knitr, rmarkdown, testthat (>= 3.0.0), S4Vectors, SummarizedExperiment

Config/testthat/edition 3

Encoding UTF-8

Roxygen list(markdown = TRUE)

RoxygenNote 7.3.3

URL https://glycoverse.github.io/glyexp/, https://github.com/glycoverse/glyexp

Imports checkmate, cli, dplyr, glue, glyrepr (>= 0.10.0), purrr, rlang, stringr, tibble, tidyr, tidyselect, vctrs

VignetteBuilder knitr

Depends R (>= 4.1)

LazyData true

BugReports https://github.com/glycoverse/glyexp/issues

Config/pak/sysreqs libglpk-dev libicu-dev libxml2-dev

Repository https://glycoverse.r-universe.dev

Date/Publication 2026-03-29 02:11:08 UTC

RemoteUrl https://github.com/glycoverse/glyexp

RemoteRef v0.14.1

RemoteSha 3e3506a9b85487b546cf43aca084efc79ed6175b

Contents

`[.glyexp_experiment` *Subsetting experiments*

Description

Getting a subset of an experiment object. Subsetting is first done on the expression matrix, then the sample information and variable information tibbles are filtered and ordered accordingly.

Syntax for `[` is similar to subsetting a matrix, with some differences:

- Both row and column indices are required, i.e. `exp[i]` is not allowed, but `exp[i, ]` and `exp[, j]` are allowed.
- `drop` argument is not supported. Subsetting an experiment always returns an new experiment, even if it has only one sample or one variable.
- Renaming the subsetted experiment is no longer supported.

Assigning to a subset of an experiment is not allowed, i.e., `exp[1, 1[ <- 0` will raise an error. You can create a new experiment with new data if needed.

Usage

```
## S3 method for class 'glyexp_experiment'
x[i, j, ...]

## S3 replacement method for class 'glyexp_experiment'
x[i, j, ...] <- value
```

Arguments

`x`	An `experiment()`.
`i, j`	Row (variable) and column (sample) indices to subset.
`...`	Ignored.
`value`	Ignored.

Value

An `experiment()` object.

Examples

```
# Create a toy experiment for demonstration
exp <- toy_experiment

# Subsetting single samples
exp[, "S1"]
exp[, 1]

# Subsetting single variables
```

```
exp["V1", ]
exp[1, ]

# Subsetting multiple samples and variables
exp[c("V1", "V2"), c("S2", "S3")]
exp[c(1, 2), c(2, 3)]

# Create a copy
exp[, ]
```

`arrange_obs` *Arrange sample or variable information*

Description

Arrange the sample or variable information tibble of an `experiment()`.

The same syntax as `dplyr::arrange()` is used. For example, to arrange samples by the "group" column, use `arrange_obs(exp, group)`. This actually calls `dplyr::arrange()` on the sample information tibble with the `group` column, and then updates the expression matrix accordingly to match the new order.

Usage

```
arrange_obs(exp, ...)

arrange_var(exp, ...)
```

Arguments

`exp`	An `experiment()`.
`...`	<`data-masking`> Variables to arrange by, passed to `dplyr::arrange()` internally.

Value

An new `experiment()` object.

Examples

```
# Create a toy experiment for demonstration
exp <- toy_experiment |>
  mutate_var(type = c("Y", "X", "Z", "Y"))

# Arrange samples by group column
arranged_exp <- arrange_obs(exp, group)
get_sample_info(arranged_exp)
get_expr_mat(arranged_exp)
```

```
# Arrange variables by type column
arranged_exp <- arrange_var(exp, type)
get_var_info(arranged_exp)
get_expr_mat(arranged_exp)

# Arrange by multiple columns
arrange_obs(exp, group, sample)
get_sample_info(arranged_exp)
get_expr_mat(arranged_exp)
```

`as_pseudo_glycome` *Convert a glycoproteomics experiment to a pseudo-glycome experiment*

Description

Transforms a glycoproteomics-type experiment into a glycomics-type experiment by aggregating expression values by glycan structure (if available) or glycan composition.

This function implements the "pseudo-glycome" method described in doi:10.1038/s4146702668579-x, which aggregates glycoproteomic data by glycans to simulate glycome data when real glycome is unavailable.

Usage

```
as_pseudo_glycome(exp, aggr_method = c("sum", "mean", "median"))
```

Arguments

`exp`	A glycoproteomics `experiment()`.
`aggr_method`	Aggregation method to use. One of "sum", "mean", or "median". Default is "sum". Note that glycopeptides can have different ionization efficiencies, so none of these methods are technically rigorous.

Details

Aggregation behavior:

- If `glycan_structure` column exists in `var_info`, aggregation is done by glycan structure (more specific)
- Otherwise, aggregation is done by `glycan_composition`
- Expression values are aggregated within each glycan group using the specified `aggr_method`

Limitation: Glycopeptides can have different ionization efficiencies, so the aggregation operation is not technically rigorous regardless of the method used. Use results with caution.

Value

A glycomics-type `experiment()` with aggregated expression values. The `var_info` will contain only `glycan_composition` and `glycan_structure` (if present in input) columns.

Examples

```
library(glyrepr)
as_pseudo_glycome(real_experiment)
```

---

as_se *Convert experiment to SummarizedExperiment*

Description

Convert an `experiment()` object to a `SummarizedExperiment` object. This function maps the experiment structure to SummarizedExperiment format:

- `expr_mat` becomes the main assay
- `sample_info` becomes `colData`
- `var_info` becomes `rowData`
- `meta_data` becomes `metadata`

Usage

```
as_se(exp, assay_name = "counts")
```

Arguments

`exp`	An `experiment()` object to convert.
`assay_name`	Character string specifying the name for the assay. Default is "counts".

Value

A `SummarizedExperiment` object.

Examples

```
# Convert toy experiment to SummarizedExperiment
se <- as_se(toy_experiment)
se
```

`as_tibble.glyexp_experiment` *Convert an experiment to a tibble*

Description

Convert an experiment object to a tibble of "tidy" format. That is, each row is a unique combination of "sample" and "variable", with the observation (the abundance) in the "value" column. Additional columns in the sample and variable information are included. This format is also known as the "long" format.

Usually you don't want all columns in the sample information or variable information tibbles to be included in the output tibble, as this will make the output tibble very "wide". You can specify which columns to include in the output tibble by passing the column names to the `sample_cols` and `var_cols` arguments. <data-masking> syntax is used here. By default, all columns are included.

Usage

```
## S3 method for class 'glyexp_experiment'
as_tibble(
  x,
  sample_cols = tidyselect::everything(),
  var_cols = tidyselect::everything(),
  ...
)
```

Arguments

`x`	An `experiment()`.
`sample_cols`	<data-masking> Columns to include from the sample information tibble.
`var_cols`	<data-masking> Columns to include from the variable information tibble.
`...`	Ignored.

Value

A tibble.

Examples

```
library(tibble)

# Create a toy experiment for demonstration
toy_exp <- toy_experiment
toy_exp

# Convert the experiment to a tibble
as_tibble(toy_exp)
```

```
# specify columns to include
as_tibble(toy_exp, sample_cols = group, var_cols = c(protein, peptide))
```

`dim.glyexp_experiment` *Dimensions of an experiment*

Description

Retrieve the dimensions of an experiment object, i.e. the number of variables and samples.

Usage

```
## S3 method for class 'glyexp_experiment'
dim(x)

## S3 replacement method for class 'glyexp_experiment'
dim(x) <- value
```

Arguments

`x`	An experiment object.
`value`	Ignored.

Value

A vector with two elements: the number of variables and the number of samples.

Examples

```
dim(real_experiment)
```

`dimnames.glyexp_experiment`
Dimname for experiment

Description

The dimnames method for `experiment()` objects are the dimnames of their expression matrix.

Usage

```
## S3 method for class 'glyexp_experiment'
dimnames(x, ...)
```

Arguments

x	An experiment().
...	Ignored.

Value

A list with the dimnames of the expression matrix.

Examples

```
dimnames(real_experiment)
```

---

experiment *Create a new experiment*

Description

The data container of a glycoproteomics or glycomics experiment. Expression matrix, sample information, and variable information are required then will be managed by the experiment object. It acts as the data core of the glycoverse ecosystem.

Usage

```
experiment(
  expr_mat,
  sample_info = NULL,
  var_info = NULL,
  exp_type = "others",
  glycan_type = NULL,
  coerce_col_types = TRUE,
  check_col_types = TRUE,
  ...
)

is_experiment(x)
```

Arguments

expr_mat	An expression matrix with samples as columns and variables as rows.
sample_info	A tibble with a column named "sample", and other columns other useful information about samples, e.g. group, batch, sex, age, etc. If NULL (default), a tibble with only one column named "sample" will be created, same as the column names of expr_mat.

`var_info` A tibble with a column named "variable", and other columns other useful information about variables, e.g. protein name, peptide, glycan composition, etc. If NULL (default), a tibble with only one column named "variable" will be created, same as the row names of `expr_mat`. Must be provided if `exp_type` is not "others".

`exp_type` The type of the experiment, "glycomics", "glycoproteomics", "traitomics", "traitproteomics", or "others". Default to "others".

`glycan_type` The type of glycan. One of "N", "O-GalNAc", "O-GlcNAc", "O-Man", "O-Fuc", "O-Glc". Can also be NULL if `exp_type` is "others".

`coerce_col_types` If common column types are coerced. Default to TRUE. If TRUE, all columns in the "Column conventions" section will be coerced to the expected types. Skipped for "others" type even if TRUE.

`check_col_types` If column type conventions are checked. Default to TRUE. Type checking is performed after column coercion (if `coerce_col_types` is TRUE). Skipped for "others" type even if TRUE.

`...` Other meta data about the experiment.

`x` An object to check.

Value

A `experiment()`. If the input data is wrong, an error will be raised.

A logical value.

Requirements of the input data

Expression matrix:

- Must be a numeric matrix with **variables as rows** and **samples as columns**.
- The **column names** must correspond to sample IDs.
- The **row names** must correspond to variable IDs.

Sample information (`sample_info`):

- Must be a tibble with a column named "sample" (sample ID).
- Each value in "sample" must be unique.
- The set of "sample" values must match the column names of the expression matrix (order does not matter).

Variable information (`var_info`):

- Must be a tibble with a column named "variable" (variable ID).
- Each value in "variable" must be unique.
- The set of "variable" values must match the row names of the expression matrix (order does not matter).

The function will automatically reorder the expression matrix to match the order of "sample" and "variable" in the info tables.

Column requirements

Some columns are required compulsorily in the variable information tibble for a valid experiment. It depends on the experiment type.

- For "glycomics": `glycan_composition`.
- For "glycoproteomics": `protein`, `protein_site`, `glycan_composition`.
- For "traitomics": no required columns.
- For "traitproteomics": `protein`, `protein_site`.
- For "others": no required columns.

See the "Column conventions" section for detailed description of these columns.

The last two types of experiments are created by the `glydet` package. Normally you don't need to manually create them.

Column conventions

`glycoverse` has some conserved column names for `sample_info` and `var_info` to make everything work seamlessly. It's not mandatory, but following these conventions will make your life easier.

sample_info:

- `group`: factor, treatment/condition/grouping, used by many `glystats` and `glyvis` functions.
- `batch`: factor, batch information, used by `glyclean::correct_batch_effect()`.
- `bio_rep`: factor, biological replicate, may be used in the future.

var_info:

- `protein`: character, protein Uniprot accession.
- `protein_site`: integer, glycosylation site position on protein.
- `gene`: character, gene symbol.
- `peptide`: character, peptide sequence.
- `peptide_site`: integer, glycosylation site position on peptide.
- `glycan_composition`: `glyrepr::glycan_composition()`, glycan composition.
- `glycan_structure`: `glyrepr::glycan_structure()`, glycan structure.

Meta data

Other meta data can be added to the `meta_data` attribute. `meta_data` is a list of additional information about the experiment. Two meta data fields are required:

- `exp_type`: "glycomics", "glycoproteomics", "traitomics", "traitproteomics", or "others"
- `glycan_type`: "N", "O-GalNAc", "O-GlcNAc", "O-Man", "O-Fuc", "O-Glc", or NULL

Other meta data will be added by other `glycoverse` packages for their own purposes.

Index columns

The **index columns** are the backbone that keep your data synchronized:

- The "sample" column in `sample_info` must match the column names of `expr_mat`.
- The "variable" column in `var_info` must match the row names of `expr_mat`.

These columns act as unique identifiers, ensuring that your expression matrix, sample information, and variable information always stay in sync, no matter how you filter, arrange, or subset your data.

Examples

```
# The minimum required input is an expression matrix.
expr_mat <- matrix(runif(9), nrow = 3, ncol = 3)
colnames(expr_mat) <- c("S1", "S2", "S3")
rownames(expr_mat) <- c("V1", "V2", "V3")
experiment(expr_mat)

# Or with more detailed information.
sample_info <- tibble::tibble(sample = c("S1", "S2", "S3"), group = c("A", "B", "A"))
var_info <- tibble::tibble(
  variable = c("V1", "V2", "V3"),
  glycan_composition = glyrepr::glycan_composition(c(Hex = 1))
)
experiment(expr_mat, sample_info, var_info, exp_type = "glycomics", glycan_type = "N")
```

`filter_obs` *Filter samples or variables of an experiment*

Description

Getting a subset of an `experiment()` by filtering samples or variables.

The same syntax as `dplyr::filter()` is used. For example, to get a subset of an experiment keeping only "HC" samples, use `filter_obs(exp, group == "HC")`. This actually calls `dplyr::filter()` on the sample information tibble with condition `group == "HC"`, and then updates the expression matrix accordingly.

Usage

```
filter_obs(exp, ..., .drop_levels = TRUE)

filter_var(exp, ..., .drop_levels = TRUE)
```

Arguments

`exp`	An `experiment()`.
`...`	<`data-masking`> Expression to filter samples or variables. passed to `dplyr::filter()` internally.
`.drop_levels`	Logical. If `TRUE`, drop unused factor levels for columns referenced in the filtering expressions.

Details

One difference between `filter_obs()` or `filter_var()` and dplyr::filter is that, when filtering on factor columns, the unused levels are automatically dropped by default. This behavior can be turnt off by setting `.drop_levels` to FALSE.

Value

An new `experiment()` object.

Examples

```
# Create a toy experiment for demonstration
exp <- toy_experiment |>
  mutate_var(type = c("X", "X", "Y", "Y"))

# Filter samples
sub_exp_1 <- filter_obs(exp, group == "A")
get_sample_info(sub_exp_1)
get_expr_mat(sub_exp_1)

# Filter variables
sub_exp_2 <- filter_var(exp, type == "X")
get_var_info(sub_exp_2)
get_expr_mat(sub_exp_2)

# Use pipe
sub_exp_3 <- exp |>
  filter_obs(group == "A") |>
  filter_var(type == "X")
get_sample_info(sub_exp_3)
get_var_info(sub_exp_3)
get_expr_mat(sub_exp_3)
```

---

from_se *Convert SummarizedExperiment to experiment*

Description

Convert a `SummarizedExperiment` object to an `experiment()` object. This function maps the SummarizedExperiment structure to experiment format:

- The main assay becomes `expr_mat`
- `colData` becomes `sample_info`
- `rowData` becomes `var_info`
- `metadata` becomes `meta_data`

Usage

```
from_se(se, assay_name = NULL, exp_type = NULL, glycan_type = NULL)
```

Arguments

se	A SummarizedExperiment object to convert.
assay_name	Character string specifying which assay to use. If NULL (default), uses the first assay.
exp_type	Character string specifying experiment type. Must be either "glycomics", "glycoproteomics", or "others". If NULL, will try to extract from metadata, otherwise defaults to "glycomics".
glycan_type	Character string specifying glycan type. Must be either "N", "O-GalNAc", "O-GlcNAc", "O-Man", "O-Fuc", or "O-Glc". If NULL, will try to extract from metadata, otherwise defaults to "N".

Value

An experiment() object.

Examples

```
# Convert SummarizedExperiment back to experiment
se <- as_se(toy_experiment)
exp_back <- from_se(se, exp_type = "glycomics", glycan_type = "N")
exp_back
```

---

get_expr_mat *Get the expression matrix of an experiment*

Description

A matrix of expression values with samples as columns and variables as rows.

Usage

```
get_expr_mat(exp)
```

Arguments

exp	An experiment().

Value

A matrix of expression values.

Examples

```
get_expr_mat(real_experiment)[1:5, 1:5]
```

get_meta_data *Get the meta data of an experiment*

Description

Meta data is some descriptions about the experiment, like the experiment type ("glycomics" or "glycoproteomics"), or the glycan type ("N", "O-GalNAc", "O-GlcNAc", "O-Man", "O-Fuc", or "O-Glc").

Usage

```
get_meta_data(exp, x = NULL)

get_exp_type(exp)

get_glycan_type(exp)
```

Arguments

exp	An experiment().
x	A string, the name of the meta data field. If NULL (default), a list of all meta data fields will be returned.

Value

The value of the meta data field. If the field does not exist, NULL will be returned.

Examples

```
get_meta_data(real_experiment)
get_exp_type(real_experiment)
get_glycan_type(real_experiment)
```

get_sample_info *Get the sample information of an experiment*

Description

A tibble of sample information, with the first column being "sample".

Usage

```
get_sample_info(exp)
```

Arguments

exp	An experiment().

Value

A tibble of sample information.

Examples

```
get_sample_info(real_experiment)
```

`get_var_info` *Get the variable information of an experiment*

Description

A `tibble` of variable information, with the first column being "variable".

Usage

```
get_var_info(exp)
```

Arguments

`exp`	An `experiment()`.

Value

A tibble of variable information.

Examples

```
get_var_info(real_experiment)
```

`left_join_obs` *Join data to sample or variable information*

Description

These functions allow you to join additional data to the sample information or variable information of an `experiment()`. They work similarly to `dplyr::left_join()`, `dplyr::inner_join()`, `dplyr::semi_join()`, and `dplyr::anti_join()`, but are designed to work with experiment objects.

After joining, the `expr_mat` is automatically updated to reflect any changes in the number of samples or variables.

Important Notes:

- The `relationship` parameter is locked to "many-to-one" to ensure that the number of observations never increases, which would violate the experiment object assumptions.
- `right_join()` and `full_join()` are not supported as they could add new observations to the experiment.

Usage

```
left_join_obs(exp, y, by = NULL, ...)

inner_join_obs(exp, y, by = NULL, ...)

semi_join_obs(exp, y, by = NULL, ...)

anti_join_obs(exp, y, by = NULL, ...)

left_join_var(exp, y, by = NULL, ...)

inner_join_var(exp, y, by = NULL, ...)

semi_join_var(exp, y, by = NULL, ...)

anti_join_var(exp, y, by = NULL, ...)
```

Arguments

exp	An `experiment()`.
y	A data frame to join to `sample_info` or `var_info`.
by	A join specification created with `dplyr::join_by()`, or a character vector of variables to join by. See `dplyr::left_join()` for details.
...	Other arguments passed to the underlying dplyr join function, except `relationship` which is locked to "many-to-one".

Value

A new `experiment()` object with updated sample or variable information.

Examples

```
library(dplyr)
library(tibble)

# Create a toy experiment
exp <- toy_experiment

# Create additional sample information to join
extra_sample_info <- tibble(
  sample = c("S1", "S2", "S3", "S4"),
  age = c(25, 30, 35, 40),
  treatment = c("A", "B", "A", "B")
)

# Left join to sample information
exp_with_extra <- left_join_obs(exp, extra_sample_info, by = "sample")
get_sample_info(exp_with_extra)
```

```
# Inner join (only keeps matching samples)
exp_inner <- inner_join_obs(exp, extra_sample_info, by = "sample")
get_sample_info(exp_inner)
get_expr_mat(exp_inner) # Note: expr_mat is updated too

# Create additional variable information to join
extra_var_info <- tibble(
  protein = c("P1", "P2", "P3"),
  pathway = c("A", "B", "A"),
  importance = c(0.8, 0.6, 0.9)
)

# Left join to variable information
exp_with_var_extra <- left_join_var(exp, extra_var_info, by = "protein")
get_var_info(exp_with_var_extra)
```

`merge.glyexp_experiment` *Merge two experiments*

Description

Merges two `experiment()` objects into a single object.

Expression matrix: Expression matrices are merged by matching variables. For variables that are present in only one of the experiments, the corresponding values in the expression matrix are set to `NA`.

Sample information: Column names and column types must be identical (order does not matter). No overlap is allowed for sample names.

Variable information: Column names and column types must be identical (order does not matter). Variable names are ignored. The identity of variables is determined by all other columns in the variable information. If row A in the first `var_info` is identical (order does not matter) to row B in the second `var_info`, they are considered the same variable. Therefore, please make sure that each row has a unique combination of values (except for `variable`), otherwise the function cannot determine which variable in `x` is identical to which variable in `y`. To ensure uniqueness, you can use `glyclean::aggregate()`.

Metadata: Metadata is taken from the first experiment. This is the only place where the order of `x` and `y` matters.

Usage

```
## S3 method for class 'glyexp_experiment'
merge(x, y, ...)
```

Arguments

`x, y`	`experiment()` objects to merge
`...`	Not used

Value

A new `experiment()` object

Variable matching

In the variable information tibble, the `variable` column is just arbitrary unique identifier, while the real identity of a variable is determined by all other columns. For example, if the variable information tibble has the following columns:

```
variable glycan_composition protein protein_site
V1       Hex(2)HexNAc(1)    P1      2
V2       Hex(2)HexNAc(1)    P2      3
V3       Hex(2)HexNAc(2)    P2      3
V4       Hex(2)HexNAc(2)    P2      3
```

We know that V1, V2, and V3 are all different variables, but V3 and V4 are the same variable (they have the same glycan composition, protein, and protein site).

During the merge, the function will use all columns in the variable information tibble except for `variable` to match variables. This means that if the combination of all other columns is not unique, the function cannot determine the identity of the variables.

To ensure uniqueness, you can use `glyclean::aggregate()`, which merges the expression levels of variables with the same identity.

After the merge, a new `variable` column is added to the variable information tibble, and used as the rownames of the expression matrix.

Variable and sample orders

The order of variables and samples in the merged experiment is deterministic.

For samples, as no overlapping is allowed for sample names, the new sample names are the union of the sample names in `x` and `y`, orders reserved.

For variables, the new variables are:

1. First, all variables in `x`, with the same order as in `x`.
2. Then, all variables in `y` while not in `x`, with the same order as in `y`.

Note that for variables, we refer to the identity of variables, not the variable names in the `variable` column.

Examples

```
# Merging is most useful with experiments from different batches.
# Here we just demonstrate the usage.

# Create experiments to be merged
exp1 <- toy_experiment
exp2 <- toy_experiment |>
  mutate_obs(sample = paste0("S", 7:12))
exp3 <- toy_experiment |>
```

```
  mutate_obs(sample = paste0("S", 13:18))

# Merge two experiments
merge(exp1, exp2)

# Merge multiple experiments
Reduce(merge, list(exp1, exp2, exp3))
```

`mutate_obs` *Mutate sample or variable information*

Description

Mutate the sample or variable information tibble of an `experiment()`.

The same syntax as `dplyr::mutate()` is used. For example, to add a new column to the sample information tibble, use `mutate_obs(exp, new_column = value)`. This actually calls `dplyr::mutate()` on the sample information tibble with `new_column = value`.

If the `sample` column in `sample_info` or the `variable` column in `var_info` is to be modified, the new column must be unique, otherwise an error is thrown. The column names or row names of `expr_mat` will be updated accordingly.

Usage

```
mutate_obs(exp, ...)

mutate_var(exp, ...)
```

Arguments

`exp`	An `experiment()`.
`...`	<`data-masking`> Name-value pairs, passed to `dplyr::mutate()` internally.

Value

An new `experiment()` object.

Examples

```
# Create a toy experiment for demonstration
exp <- toy_experiment |>
  mutate_var(type = c("X", "X", "Y", "Y"))

# Add a new column to sample information tibble or variable information tibble
exp |>
  mutate_obs(new_column = c(1, 2, 3, 4, 5, 6)) |>
  get_sample_info()
```

```
exp |>
  mutate_var(new_column = c("A", "A", "B", "B")) |>
  get_var_info()

# Modify existing columns
exp |>
  mutate_obs(group = dplyr::if_else(group == "A", "good", "bad")) |>
  get_sample_info()

exp |>
  mutate_var(type = dplyr::if_else(type == "X", "good", "bad")) |>
  get_var_info()

# Modify the `sample` column in sample information tibble
new_exp <- mutate_obs(exp, sample = c("SI", "SII", "SIII", "SIV", "SV", "SVI"))
get_sample_info(new_exp)
get_expr_mat(new_exp)

# Modify the `variable` column in variable information tibble
new_exp <- mutate_var(exp, variable = c("VI", "VII", "VIII", "VIV"))
get_var_info(new_exp)
get_expr_mat(new_exp)
```

`n_samples` *Get number of samples or variables of an experiment*

Description

Getting the number of samples or variables of an `experiment()`. Syntax sugar for `ncol(exp$expr_mat)` and `nrow(exp$expr_mat)`.

Usage

```
n_samples(exp)

n_variables(exp)
```

Arguments

`exp`	An `experiment()`.

Value

An integer with the number of samples or variables.

Examples

```
exp <- toy_experiment
n_samples(exp)
n_variables(exp)
```

`real_experiment` *Real glycoproteomics experiment*

Description

A real glycoproteomics experiment object. This dataset is derived from the unpublished ProteomeXchange dataset PXD063749. It contains serum N-glycoproteome profiles from 12 patients with different liver conditions: healthy (H), hepatitis (M), cirrhosis (Y), and hepatocellular carcinoma (C). Glycopeptide identification and annotation were performed using pGlyco3 (https://github.com/pFindStudio/pGlyco
and quantification was carried out with pGlycoQuant (https://github.com/Power-Quant/pGlycoQuant).
The raw data were imported with `glyread::read_pglyco3_pglycoquant()`.

Usage

```
real_experiment
```

Format

An `experiment()` object with 4262 variables and 12 samples.

Variable information::

- `peptide`: peptide sequence
- `peptide_site`: site position on the peptide
- `protein`: protein accession
- `protein_site`: site position on the protein
- `gene`: gene symbol
- `glycan_composition`: glycan composition (glyrepr::glycan_composition())
- `glycan_structure`: glycan structure (glyrepr::glycan_structure())

Sample information::

- `sample`: sample ID
- `group`: disease group, one of "H" (healthy), "M" (hepatitis), "Y" (cirrhosis), and "C" (hepatocellular carcinoma)

real_experiment2 *Real glycomics experiment*

Description

A real glycomics experiment object. This dataset is derived from the unpublished glycoPOST dataset GPST000313. It contains serum N-glycome profiles from 144 samples with different liver conditions: healthy (H), hepatitis (M), cirrhosis (Y), and hepatocellular carcinoma (C).

Usage

```
real_experiment2
```

Format

An `experiment()` object with 67 variables and 144 samples.

Variable information::

- `glycan_composition`: glycan composition (glyrepr::glycan_composition())
- `glycan_structure`: glycan structure (glyrepr::glycan_structure())

Sample information::

- `sample`: sample ID
- `group`: disease group, one of "H" (healthy), "M" (hepatitis), "Y" (cirrhosis), and "C" (hepatocellular carcinoma)

rename_obs *Rename columns in the sample or variable information tibble*

Description

These two functions provide a way to rename columns in the sample or variable information tibble of an `experiment()`.

The same syntax as `dplyr::rename()` is used. For example, to rename the "group" column in the sample information tibble to "condition", use `rename_obs(exp, condition = group)`. Note that you can't rename the "sample" column in the sample information tibble, as well as the "variable" column in the variable information tibble. These two columns are used to link the sample or variable information tibble to the expression matrix.

Usage

```
rename_obs(exp, ...)

rename_var(exp, ...)
```

Arguments

exp	An experiment().
...	<data-masking> Name pairs to rename. Use new_name = old_name to rename columns.

Value

An new experiment() object.

Examples

```
toy_exp <- toy_experiment
toy_exp

# Rename columns in sample information tibble
rename_obs(toy_exp, condition = group)

# Rename columns in variable information tibble
rename_var(toy_exp, composition = glycan_composition)
```

---

samples *Get Samples or Variables of an Experiment*

Description

Getting the names of samples or variables of an experiment(). Syntax sugar for colnames(exp$expr_mat) and rownames(exp$expr_mat).

Usage

```
samples(exp)

variables(exp)
```

Arguments

exp	An experiment().

Value

A character vector of sample or variable names.

Examples

```
exp <- toy_experiment
samples(exp)
variables(exp)
```

`select_obs` *Select columns of the sample or variable information tibble*

Description

These two functions provide a way to trimming down the sample or variable information tibble of an `experiment()` to only the columns of interest.

The same syntax as `dplyr::select()` is used. For example, to get a new `experiment()` with only the "sample" and "group" columns in the sample information tibble, use `select_obs(exp, group)`. Note that you don't need to (and you can't) explicitly select or deselect the `sample` column in `sample_info`. The same applies to the `variable` column in `var_info`. Whatever the selection expression is, the `sample` or `variable` column will always be kept.

Usage

```
select_obs(exp, ...)

select_var(exp, ...)
```

Arguments

`exp`	An `experiment()`.
`...`	<`data-masking`> Column names to select. If empty, all columns except the `sample` or `variable` column will be discarded.

Details

When using `select_var()` with `dplyr`, you may encounter package conflicts. `dplyr` also has a function called `select_var()` that has been deprecated for over two years. If you encounter package conflicts, use the following code to resolve them:

```
conflicted::conflicts_prefer(glyexp::select_var)
```

Value

An new `experiment()` object.

Examples

```
toy_exp <- toy_experiment

toy_exp_2 <- toy_exp |>
  select_obs(group) |>
  select_var(protein, peptide)

get_sample_info(toy_exp_2)
get_var_info(toy_exp_2)
```

set_meta_data *Set the meta data of an experiment*

Description

Set meta data values for the experiment, like the experiment type ("glycomics" or "glycoproteomics"), or the glycan type ("N", "O-GalNAc", "O-GlcNAc", "O-Man", "O-Fuc", or "O-Glc").

Usage

```
set_meta_data(exp, x, value)

set_exp_type(exp, value)

set_glycan_type(exp, value)
```

Arguments

exp	An experiment().
x	A string, the name of the meta data field.
value	The value to set for the meta data field.

Value

The modified experiment() object.

slice_obs *Slice sample or variable information*

Description

Slice the sample or variable information tibble of an experiment().

These functions provide row-wise slicing operations similar to dplyr's slice functions. They select rows by position or based on values in specified columns, and update the expression matrix accordingly to match the new selection.

- slice_obs() and slice_var(): Select rows by position
- slice_head_obs() and slice_head_var(): Select first n rows
- slice_tail_obs() and slice_tail_var(): Select last n rows
- slice_sample_obs() and slice_sample_var(): Select random n rows
- slice_max_obs() and slice_max_var(): Select rows with highest values
- slice_min_obs() and slice_min_var(): Select rows with lowest values

Usage

```
slice_obs(exp, ...)

slice_var(exp, ...)

slice_head_obs(exp, n, prop)

slice_head_var(exp, n, prop)

slice_tail_obs(exp, n, prop)

slice_tail_var(exp, n, prop)

slice_sample_obs(exp, n, prop, weight_by = NULL, replace = FALSE)

slice_sample_var(exp, n, prop, weight_by = NULL, replace = FALSE)

slice_max_obs(exp, order_by, ..., n, prop, with_ties = TRUE, na_rm = FALSE)

slice_max_var(exp, order_by, ..., n, prop, with_ties = TRUE, na_rm = FALSE)

slice_min_obs(exp, order_by, ..., n, prop, with_ties = TRUE, na_rm = FALSE)

slice_min_var(exp, order_by, ..., n, prop, with_ties = TRUE, na_rm = FALSE)
```

Arguments

exp	An experiment().
...	<data-masking> For slice_*(), integer row positions. For slice_max() and slice_min(), variables to order by. Other arguments passed to the corresponding dplyr function.
n	For slice_head(), slice_tail(), slice_sample(), slice_max(), and slice_min(), the number of rows to select.
prop	For slice_head(), slice_tail(), slice_sample(), slice_max(), and slice_min(), the proportion of rows to select.
weight_by	For slice_sample(), sampling weights.
replace	For slice_sample(), should sampling be with replacement?
order_by	For slice_max() and slice_min(), variable to order by.
with_ties	For slice_max() and slice_min(), should ties be kept?
na_rm	For slice_max() and slice_min(), should missing values be removed?

Value

A new experiment() object.

Examples

```
# Create a toy experiment for demonstration
exp <- toy_experiment |>
  mutate_obs(score = c(10, 20, 30, 15, 25, 35)) |>
  mutate_var(value = c(5, 10, 15, 8))

# Select specific rows by position
slice_obs(exp, 1, 3, 5)

# Select first 3 samples
slice_head_obs(exp, n = 3)

# Select last 2 variables
slice_tail_var(exp, n = 2)

# Select 2 random samples
slice_sample_obs(exp, n = 2)

# Select samples with highest scores
slice_max_obs(exp, order_by = score, n = 2)

# Select variables with lowest values
slice_min_var(exp, order_by = value, n = 2)
```

---

split.glyexp_experiment

Split an experiment

Description

Devides an `experiment()` into a list of `experiment()` objects by `f`.

Usage

```
## S3 method for class 'glyexp_experiment'
split(x, f, drop = FALSE, where = "var_info", ...)
```

Arguments

`x`	An `experiment()`.
`f`	<data-masking> A column in `var_info` or `sample_info` that `as.factor(f)` defines the grouping.
`drop`	Logical indicating if levels that do not occur should be dropped. Defaults to FALSE.
`where`	Where to find the column, "var_info" or "sample_info".
`...`	Ignored

Value

A named list of `experiment()` objects.

Examples

```
split(toy_experiment, group, where = "sample_info")
```

`standardize_variable` *Standardize variable IDs in an experiment*

Description

Converts meaningless variable IDs (like "GP1", "V1") into meaningful, human-readable IDs based on the experiment type and available columns.

The format of the new IDs depends on the `exp_type` if `format` is not specified:

- `glycomics`: `{glycan_composition}`, e.g., "Hex(5)HexNAc(2)"
- `glycoproteomics`: `{protein}-{protein_site}-{glycan_composition}`
- `traitomics`: `{motif}` or `{trait}` depending on which column is present
- `traitproteomics`: `{protein}-{protein_site}-{motif}` or `{protein}-{protein_site}-{trait}`

If duplicate IDs are generated (e.g., same composition with multiple PSMs), a unique integer suffix is appended using the `unique_suffix` pattern.

Usage

```
standardize_variable(exp, format = NULL, unique_suffix = "-{N}")
```

Arguments

`exp`	An `experiment()`.
`format`	A format string specifying how to construct variable IDs. Use `{column_name}` to insert values from `var_info` columns. For example, `"{gene}-{glycan_composition}"` would produce "GENE1-Hex(5)". If `NULL` (default), a sensible format is chosen based on `exp_type`.
`unique_suffix`	A string pattern for making IDs unique when duplicates exist. Must contain `{N}` which will be replaced with the numeric suffix (1, 2, 3...). Default is `"-{N}"` which produces IDs like "Hex(5)-1", "Hex(5)-2".

Value

The experiment with standardized variable IDs.

Examples

```
# Glycomics example
expr_mat <- matrix(1:4, nrow = 2)
rownames(expr_mat) <- c("V1", "V2")
colnames(expr_mat) <- c("S1", "S2")
sample_info <- tibble::tibble(sample = c("S1", "S2"))
var_info <- tibble::tibble(
  variable = c("V1", "V2"),
  glycan_composition = glyrepr::glycan_composition(c(Hex = 5, HexNAc = 2))
)
exp <- experiment(expr_mat, sample_info, var_info,
  exp_type = "glycomics", glycan_type = "N"
)
standardize_variable(exp)

# Glycoproteomics example
expr_mat <- matrix(1:4, nrow = 2)
rownames(expr_mat) <- c("GP1", "GP2")
colnames(expr_mat) <- c("S1", "S2")
sample_info <- tibble::tibble(sample = c("S1", "S2"))
var_info <- tibble::tibble(
  variable = c("GP1", "GP2"),
  protein = c("P12345", "P12345"),
  protein_site = c(32L, 45L),
  glycan_composition = glyrepr::glycan_composition(c(Hex = 5, HexNAc = 2))
)
exp <- experiment(expr_mat, sample_info, var_info,
  exp_type = "glycoproteomics", glycan_type = "N"
)
standardize_variable(exp)

# Custom format example
standardize_variable(exp, format = "{protein}-{glycan_composition}")
```

---

summarize_experiment *Identification overview*

Description

This function summarizes the number of glycan compositions, glycan structures, glycopeptides, peptides, glycoforms, glycoproteins, and glycosites in an `experiment()`.

Usage

```
summarize_experiment(x, count_struct = NULL)
```

Arguments

x	An experiment() object.
count_struct	For counting glycopeptides and glycoforms. whether to count the number of glycan structures or glycopeptides. If TRUE, glycopeptides or glycoforms bearing different glycan structures with the same glycan composition are counted as different ones. If not provided (NULL), defaults to TRUE if glycan_structure column exists in the variable information tibble, otherwise FALSE.

Details

The following columns are required in the variable information tibble:

- composition: glycan_composition
- structure: glycan_structure
- peptide: peptide
- glycopeptide: glycan_composition or glycan_structure, peptide, peptide_site
- glycoform: glycan_composition or glycan_structure, protein or proteins, protein_site or protein_sites
- protein: protein or proteins
- glycosite: protein or proteins, protein_site or protein_sites

You can use count_struct parameter to control how to count glycopeptides and glycoforms, either by glycan structures or by glycan compositions.

Value

A tibble with columns item and n summarizing the results. The items include:

- total_composition: The number of glycan compositions.
- total_structure: The number of glycan structures.
- total_peptide: The number of peptides.
- total_glycopeptide: The number of unique combinations of peptides, sites, and glycans.
- total_glycoform: The number of unique combinations of proteins, sites, and glycans.
- total_protein: The number of proteins.
- total_glycosite: The number of unique combinations of proteins and sites.

In addition, _per_sample items are calculated as the average number of detected items per sample. For example, composition_per_sample is the average number of glycan compositions detected per sample.

Examples

```
exp <- real_experiment
summarize_experiment(exp)
```

toy_experiment *Toy experiment*

Description

A toy experiment object for new users to play with to get familiar with `experiment()` objects.

Usage

```
toy_experiment
```

Format

`toy_experiment`:
An `experiment()` object with 4 variables and 6 samples. `var_info` contains fields: `protein`, `peptide`, and `glycan_composition`. `sample_info` contains fields: `group` and `batch`.

Index