Package 'SpectriPy'

August 23, 2025

Title Enhancing Cross-Language Mass Spectrometry Data Analysis with R and Python

Version 0.99.10

Description The SpectriPy package allows integration of Python-based MS analysis code with the Spectra package. Spectra objects can be converted into Python MS data structures. In addition, SpectriPy integrates and wraps the similarity scoring and processing/filtering functions from the Python matchms package into R.

Depends R (>= 4.1.0), reticulate (>= 1.42.0)

Imports Spectra (>= 1.17.7), IRanges, S4Vectors, MsCoreUtils, ProtGenerics, methods

Suggests testthat, quarto, MsBackendMgf, MsDataHub, mzR, knitr, BiocStyle

License Artistic-2.0

BugReports https://github.com/RforMassSpectrometry/SpectriPy/issues

URL https://github.com/RforMassSpectrometry/SpectriPy

biocViews Infrastructure, Metabolomics, MassSpectrometry, Proteomics

Encoding UTF-8

SystemRequirements python (>= 3.12), pandoc, quarto

VignetteBuilder quarto

Roxygen list(markdown = TRUE)

RoxygenNote 7.3.2

Collate 'conversion.R' 'MsBackendPython.R' 'compareSpectriPy.R' 'filterSpectriPy.R' 'zzz.R'

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

git_branch devel

git_last_commit ff8cea4

git_last_commit_date 2025-08-20

Repository Bioconductor 3.22

Date/Publication 2025-08-22

Author Michael Witting [aut] (ORCID: <https://orcid.org/0000-0002-1462-4426>),
Johannes Rainer [aut, cre] (ORCID:
<https://orcid.org/0000-0002-6977-7147>),
Carolin Huber [aut] (ORCID: <https://orcid.org/0000-0002-9355-8948>),
Helge Hecht [ctb] (ORCID: <https://orcid.org/0000-0001-6744-996X>),
Marilyn De Graeve [aut] (ORCID:
<https://orcid.org/0000-0001-6916-401X>),
Wout Bittremieux [aut] (ORCID: <https://orcid.org/0000-0002-3105-1359>),
Thomas Naake [aut] (ORCID: <https://orcid.org/0000-0001-7917-5580>),
Victor Chrone [ctb] (ORCID: <https://orcid.org/0009-0007-2121-4066>),
Matthias Anagho-Mattanovich [ctb] (ORCID:
<https://orcid.org/0000-0001-7561-7898>),
Pierre Marchal [ctb] (ORCID: <https://orcid.org/0009-0006-6567-6257>),
Philippine Louail [ctb] (ORCID:
<https://orcid.org/0009-0007-5429-6846>)

Maintainer Johannes Rainer <Johannes.Rainer@eurac.edu>

Contents

`compareSpectriPy` *Spectra similarity calculations using Python's matchms library*

Description

The `compareSpectriPy()` function allows to calculate spectral similarity scores using the `calculate_scores()` function of the Python matchms.similarity. module.

Selection and configuration of the algorithm can be performed with one of the *parameter* objects/functions:

- `CosineGreedy()`: calculate the *cosine similarity score* between spectra. The score is calculated by finding the best possible matches between peaks of two spectra. Two peaks are considered a potential match if their m/z ratios lie within the given `tolerance`. The underlying peak assignment problem is here solved in a *greedy* way. This can perform notably faster, but does occasionally deviate slightly from a fully correct solution (as with the `CosineHungarian` algorithm). In practice this will rarely affect similarity scores notably, in particular for smaller tolerances. The algorithm can be configured with parameters `tolerance`, `mz_power` and `intensity_power` (see parameter description for more details). See also matchms CosineGreedy for more information.
- `CosineHungarian()`: calculate the *cosine similarity score* as with `CosineGreedy`, but using the Hungarian algorithm to find the best matching peaks between the compared spectra. The algorithm can be configured with parameters `tolerance`, `mz_power` and `intensity_power` (see parameter description for more details). See also matchms CosingHungarian for more information.

- `ModifiedCosine()`: The modified cosine score aims at quantifying the similarity between two mass spectra. The score is calculated by finding the best possible matches between peaks of two spectra. Two peaks are considered a potential match if their m/z ratios lie within the given `tolerance`, or if their m/z ratios lie within the tolerance once a mass shift is applied. The mass shift is simply the difference in precursor-m/z between the two spectra. See also matchms ModifiedCosine for more information.
- `NeutralLossesCosine()`: The neutral losses cosine score aims at quantifying the similarity between two mass spectra. The score is calculated by finding the best possible matches between peaks of two spectra. Two peaks are considered a potential match if their m/z ratios lie within the given `tolerance` once a mass shift is applied. The mass shift is the difference in precursor-m/z between the two spectra. See also matchms NeutralLossesCosine for more information.
- `FingerprintSimilarity()`: Calculate similarity between molecules based on their fingerprints. For this similarity measure to work, fingerprints are expected to be derived by running *add_fingerprint()*. See also matchms FingerprintSimilarity for more information.

Usage

```
CosineGreedy(tolerance = 0.1, mz_power = 0, intensity_power = 1)

CosineHungarian(tolerance = 0.1, mz_power = 0, intensity_power = 1)

ModifiedCosine(tolerance = 0.1, mz_power = 0, intensity_power = 1)

NeutralLossesCosine(
  tolerance = 0.1,
  mz_power = 0,
  intensity_power = 1,
  ignore_peaks_above_precursor = TRUE
)

## S4 method for signature 'Spectra,Spectra,CosineGreedy'
compareSpectriPy(x, y, param, ...)

## S4 method for signature 'Spectra,missing,CosineGreedy'
compareSpectriPy(x, y, param, ...)
```

Arguments

`tolerance`	`numeric(1)`: tolerated differences in the peaks' m/z. Peaks with m/z differences `<= tolerance` are considered matching.
`mz_power`	`numeric(1)`: the power to raise m/z to in the cosine function. The default is 0, in which case the peak intensity products will not depend on the m/z ratios.
`intensity_power`	`numeric(1)`: the power to raise intensity to in the cosine function. The default is 1.
`ignore_peaks_above_precursor`	For `NeutralLossesCosine()`: `logical(1)`: if `TRUE` (the default), peaks with m/z values larger than the precursor m/z are ignored.
`x`	A `Spectra::Spectra()` object.
`y`	A `Spectra::Spectra()` object to compare against. If missing, spectra similarities are calculated between all spectra in `x`.

`param`	One of the parameter classes listed above (such as `CosineGreedy`) defining the similarity scoring function in Python and its parameters.
`...`	ignored.

Value

`compareSpectriPy()` Returns a `numeric` matrix with the scores, with the number of rows equal to `length(x)` and the number of columns equal to `length(y)`.

Note

Parameters and algorithms are named as originally defined in the *matchms* library (i.e. all parameters in *snake_case* while *CamelCase* is used for the algorithms.

Author(s)

Carolin Huber, Michael Witting, Johannes Rainer, Helge Hecht, Marilyn De Graeve

See Also

`Spectra::compareSpectra()` in the *Spectra* package for pure R implementations of spectra similarity calculations.

Examples

```
library(Spectra)
## Create some example Spectra.
DF <- DataFrame(
    msLevel = c(2L, 2L, 2L),
    name = c("Caffeine", "Caffeine", "1-Methylhistidine"),
    precursorMz = c(195.0877, 195.0877, 170.0924)
)
DF$intensity <- list(
    c(340.0, 416, 2580, 412),
    c(388.0, 3270, 85, 54, 10111),
    c(3.407, 47.494, 3.094, 100.0, 13.240)
)
DF$mz <- list(
    c(135.0432, 138.0632, 163.0375, 195.0880),
    c(110.0710, 138.0655, 138.1057, 138.1742, 195.0864),
    c(109.2, 124.2, 124.5, 170.16, 170.52)
)
sps <- Spectra(DF)

## Calculate pairwise similarity beween all spectra within sps with
## matchms' CosineGreedy algorithm
## Note: the first compareSpectriPy will take longer because the Python
## environment needs to be set up.
res <- compareSpectriPy(sps, param = CosineGreedy())
res

## Next we calculate similarities for all spectra against the first one
res <- compareSpectriPy(sps, sps[1], param = CosineGreedy())

## Calculate pairwise similarity of all spectra in sps with matchms'
## ModifiedCosine algorithm
```

```
res <- compareSpectriPy(sps, param = ModifiedCosine())
res

## Note that the ModifiedCosine method requires the precursor m/z to be
## known for all input spectra. Thus, it is advisable to remove spectra
## without precursor m/z before using this algorithm.
sps <- sps[!is.na(precursorMz(sps))]
compareSpectriPy(sps, param = ModifiedCosine())
```

`conversion` *Converting between R and Python MS data structures*

Description

The `rspec_to_pyspec()` and `pyspec_to_rspec()` functions allow to convert (translate) MS data structures between R and Python. At present the R `Spectra::Spectra()` objects can be either translated into a list of matchms Python `matchms.Spectrum` objects or spectrum_utils Python `spectrum_utils.spectrum.MsmsSpectrum` objects. For better integration with the *reticulate* R package also a `r_to_py.Spectra()` method is available.

The mapping of spectra variables (in R) to (Python) spectra metadata can be configured and defined with the `setSpectraVariableMapping()` and `spectraVariableMapping()`. These get and set the *global* (system wide) setting and are thus also used by the `r_to_py()` method.

Properties for translation to the MS data objects of the different Python libraries are:

- *matchms*: the `matchms.Spectrum` objects support arbitrary metadata, so any spectra variable can be translated and stored in these objects.
- *spectrum_utils*: the `spectrum_utils.spectrum.MsmsSpectrum` object supports metadata variables *identifier* (`character`), *precursor_mz* (`numeric`), *precursor_charge* (`integer`) and optionally also *retention_time* (`numeric`).

See the indivudual function's documentation for more details.

Function to convert R Spectra objects into a Python list of matchms Spectrum objects using the `reticulate` package.

Usage

```
## S4 method for signature 'character'
spectraVariableMapping(object, ...)

## S4 method for signature 'missing'
spectraVariableMapping(object, ...)

setSpectraVariableMapping(x)

defaultSpectraVariableMapping()

## S3 method for class 'Spectra'
r_to_py(x, convert = FALSE)

rspec_to_pyspec(
  x,
```

```
  mapping = spectraVariableMapping(),
  pythonLibrary = c("matchms", "spectrum_utils")
)

pyspec_to_rspec(
  x,
  mapping = spectraVariableMapping(),
  pythonLibrary = c("matchms", "spectrum_utils")
)
```

Arguments

object	For spectraVariableMapping(): not used.
...	For spectraVariableMapping(): not used.
x	Spectra object.
convert	Boolean; should Python objects be automatically converted to their R equivalent? Defaults to FALSE.
mapping	named character() vector defining which spectra variables/metadata should be translated between R and Python and how they should be renamed. Defaults to spectraVariableMapping().
pythonLibrary	For rspec_to_pyspec() and pyspec_to_rspec(): character(1) defining the Python library to which (or from which) data structures the data should be converted. Possible options are "matchms" or "spectrum_utils" with "matchms" being the default.

Value

For r_to_py.Spectra() and rspec_to_pyspec(): Python list of MS data structures, either matchms.Spectrum or spectrum_utils.spectrum.MsmsSpectrum objects. For pyspec_to_rspec(): Spectra::Spectra() with the MS data of all matchms.Spectrum objects in the submitted list.

Translation of MS data objects

MS data structures can be translated between R and Python using the rspec_to_pyspec() and pyspec_to_rspec() functions, or with the r_to_py() method.

- rspec_to_pyspec() translates an R Spectra::Spectra() object into a list of Python MS data objects, which can be, depending on parameter pythonLibrary, matchms.Spectrum objects (for pythonLibrary = "matchms", the default) or spectrum_utils.spectrum.MsmsSpectrum objects (for pythonLibrary = "spectrum_utils"). Parameter mapping allows to specify which spectra variables from the Spectra object x should be converted in addition to the peaks data (m/z and intensity values). It defaults to mapping = spectraVariableMapping() (See the respective help below for more information on the variable mapping). While being fast, this function first loads all peaks and spectra data into memory before translating to Python data structures. A less memory intense operation could be to call this function in a loop to only load parts of the data at a time into memory.
- pyspec_to_rspec() translates a single, or a list of matchms.Spectrum objects (with parameter pythonLibrary = "matchms", the default) or a list of spectrum_utils.spectrum.MsmsSpectrum objects (with parameter pythonLibrary = "spectrum_utils") to a Spectra::Spectra() object. Parameter mapping allows to specify the metadata variables that should be translated and mapped in addition to the peaks data. The library used to represent the MS data in Python needs to be specified with parameter pythonLibrary.

- `r_to_py.Spectra()` is equivalent to `rspec_to_pyspec(pythonLibrary = "matchms")`. The spectra variables that should be converted can be configures with `setSpectraVariableMapping()` (see documentation below).

Mapping of spectra variables (metadata)

Metadata for MS spectra are represented and stored as *spectra variables* in the R `Spectra::Spectra()` objects. Also Python MS data structures store such metadata along with the mass peak data. While spectra metadata is thus supported by data structures in both programming languages, different names and naming conventions are used. The `spectraVariableMapping()` and `setSpectraVariableMapping()` functions allow to define how the names of spectra metadata (spectra variables) should be translated between R and Python. To support also the different naming conventions used by the Python libraries *matchms* and *spectrum_utils*, `spectraVariableMapping()` defines different mapping schemes for these, using by default the mapping for *matchms*. Note also that *spectrum_utils* supports only few selected metadata/spectra variables, so any additional spectra variables defined by the mapping will be ignored. The `r_to_py()` and `py_to_r()` functions will use the selected naming scheme to name the spectra variables accordingly. Also, only spectra metadata/variables in `spectraVariableMapping()` will be translated. The initial mapping is based on this definition in matchms.

- `defaultSpectraVariableMapping()`: returns the *default* mapping between spectra variables and Python metadata names for the *matchms* library.
- `spectraVariableMapping()`: returns the currently defined spectra variable mapping as a named character vector, with names representing the names of the spectra variables in R and elements the respective names of the spectra metadata in Python. Use `Spectra::spectraVariables()` on the `Spectra` object that should be converted with `r_to_py()` to list all available spectra variables. `r_to_py()` and `py_to_r()` for MS data structures will use this default mapping. Calling `spectraVariableMapping()` defining also the Python library (e.g., `spectraVariableMapping("matchms` or `spectraVariableMapping("spectrum_utils")`) will return the variable mapping for the specified Python library.
- `setSpectraVariableMapping()`: sets/replaces the currently defined mapping of spectra variable names to Python metadata names. Setting `setSpectraVariableMapping(character())` will only convert the mass peaks data (m/z and intensity values) but no spectra metadata.

Author(s)

Michael Witting, Johannes Rainer, Wout Bittremieux, Thomas Naake

Examples

```
## Import a MGF file as a `Spectra` object
library(MsBackendMgf)
library(SpectriPy)
s <- Spectra(
    system.file("extdata", "mgf", "spectra2.mgf", package = "SpectriPy"),
    source = MsBackendMgf())
s

#########################
## Conversion R to Python

## A `Spectra` can be translated to a `list` of `matchms.Spectrum` objects
## using either the `r_to_py()` method or the `rspec_to_pyspec()` function:
s_py <- r_to_py(s)
```

```
s_py

## The `s_py` can now be used like any other Python variable within the R
## *reticulate* framework. Below we extract the m/z values of the first
## spectrum
s_py[0]$mz

## Extracting that information from the `Spectra` object in R
s[1]$mz

## The `spectraVariableMapping()` defines which spectra variables (metadata)
## should be translated between R and Python:
spectraVariableMapping()

## The names of that character vector represent the names of the spectra
## variables in R, the elements the name of the metadata variable in Python.
## Below we list the available metadata information from the first
## Spectrum in Python
s_py[0]$metadata

## `setSpectraVariableMapping()` allows to replace the default mapping
## of variables. Below we e.g. add a new spectra variable to the `Spectra`
## object.
s$new_col <- 1:4

## To translate that variable to Python we need to include it to the
## `spectraVariableMapping()`. Below we define to translate only the
## precursor m/z and the new spectra variable to Python. Be aware that
## `setSpectraVariableMapping()` **globally** sets the default for any
## spectra variable mapping between R and Python. Thus, any subsequent
## calls mapping calls will use the same mapping. It is suggested to
## eventually *restore* the default mapping again after the call or
## use the `rspec_to_pyspec()` function instead, that allows to configure
## the mapping using a parameter `mapping`.
setSpectraVariableMapping(
    c(precursorMz = "precursor_mz", new_col = "new_col"))
s_py <- r_to_py(s)

s_py[0]$metadata

## Restoring the global spectra variable mapping configuration to
## the default mapping:
setSpectraVariableMapping(defaultSpectraVariableMapping())

## As an alternative to the `r_to_py()` we can use the `rspec_to_pyspec()`
## function and provide a custom mapping using the `mapping` parameter:
s_py <- rspec_to_pyspec(
    s, mapping = c(precursorMz = "precursor_mz", new_col = "new_col"))

## Convert to MS data objects from the spectrum_utils Python library
s_py2 <- rspec_to_pyspec(
    s, mapping = spectraVariableMapping("spectrum_utils"),
    pythonLibrary = "spectrum_utils")

## Convert the data back to R
pyspec_to_rspec(s_py2, pythonLibrary = "spectrum_utils")
```

```
########################
## Conversion Python to R

## A `list` of `matchms.Spectrum` objects in Python can be translated into
## the corresponding MS data structure in R (i.e. a `Spectra`) object using
## the `pyspec_to_rspec()` function:
res <- pyspec_to_rspec(s_py)
res

## All spectra from Python are thus converted into a single `Spectra` object.

## Or providing a custom variable mapping:
res <- pyspec_to_rspec(
    s_py, mapping = c(precursorMz = "precursor_mz", new_col = "new_col"))
res$new_col
```

`filterSpectriPy` *Filter Spectra using Python's matchms library*

Description

The `filterSpectriPy()` function allows to filter/process a `Spectra` object using the `select_by_intensity()`, `select_by_mz()`, `remove_peaks_around_precursor_mz()`, and `normalize_intensities()` functions of the Python matchms.filtering module.

Selection and configuration of the algorithm can be performed with one of the parameter objects (equivalent to *matchms*' function names):

- `select_by_intensity()`: Keeps only the peaks within defined intensity range (keep if `intensity_from` >= intensity >= `intensity_to`). See also the respective documentation in *matchms*.
- `select_by_mz()`: Keeps only the peaks between `mz_from` and `mz_to` (keep if `mz_from` >= m/z >= `mz_to`). See also the respective documentation in *matchms*.
- `remove_peaks_around_precursor_mz()`: Removes the peaks that are within `mz_tolerance` (in Da) of the precursor mz, excluding the precursor peak.
- `normalize_intensities()`: Normalizes the intensities of peaks (and losses) to unit height.

Usage

```
select_by_intensity(intensity_from = 10, intensity_to = 200)

select_by_mz(mz_from = 0, mz_to = 1000)

remove_peaks_around_precursor_mz(mz_tolerance = 17)

normalize_intensities()

## S4 method for signature 'Spectra,filter_param'
filterSpectriPy(object, param, mapping = spectraVariableMapping(), ...)
```

Arguments

intensity_from	numeric(1): Set lower threshold for peak intensity. Default is 10.
intensity_to	numeric(1): Set upper threshold for peak intensity. Default is 200.
mz_from	numeric(1): Set lower threshold for m/z peak positions. Default is 0.
mz_to	numeric(1): Set upper threshold for m/z peak positions. Default is 1000.
mz_tolerance	numeric(1): Tolerance of m/z values that are not allowed to lie within the precursor mz. Default is 17 Da.
object	A Spectra::Spectra() object.
param	one of parameter classes listed above (such as select_by_intensity()) defining the filter/processing function in Python and its parameters.
mapping	named character() defining which spectra variables/metadata should be converted between R and Python and how they should be renamed. Defaults to spectraVariableMapping(). See setSpectraVariableMapping() for more information.
...	ignored.

Value

filterSpectriPy() returns a Spectra object on which the filtering/processing function has been applied

Note

The first call to the filterSpectriPy() function can take longer because the Python environment needs to be first set up.

filterSpectriPy() first translates the Spectra to Python, applies the filter functions from the *matchms* Python libraries and then translates the filtered data back to a Spectra object. Thus, any spectra variables other than those that are translated between R and Python will be lost during the processing. Use setSpectraVariableMapping() to define which spectra variables should be transferred/converted between R and Python. See also examples below for more information.

The Spectra::Spectra() object returned by filterSpectriPy() will **always** use an in-memory backend (i.e. the Spectra::MsBackendMemory()), independently of the backend used by the backend used by the input Spectra.

Author(s)

Thomas Naake

See Also

- Spectra::filterIntensity(), Spectra::filterMzRange(), Spectra::scalePeaks() in the Spectra package for pure R implementations of filtering/processing calculations.
- rspec_to_pyspec() or pyspec_to_rspec() for the functions used to translated the MS data between R and Python.

Examples

```
library(Spectra)

## create some example Spectra
DF <- DataFrame(
    msLevel = c(2L, 2L, 2L),
    name = c("Caffeine", "Caffeine", "1-Methylhistidine"),
    precursorMz = c(195.0877, 195.0877, 170.0924)
)
DF$intensity <- list(
    c(340.0, 416, 2580, 412),
    c(388.0, 3270, 85, 54, 10111),
    c(3.407, 47.494, 3.094, 100.0, 13.240))
DF$mz <- list(
    c(135.0432, 138.0632, 163.0375, 195.0880),
    c(110.0710, 138.0655, 138.1057, 138.1742, 195.0864),
    c(109.2, 124.2, 124.5, 170.16, 170.52))
sps <- Spectra(DF)

## Filter: select_by_intensity
res <- filterSpectriPy(
    sps, select_by_intensity(intensity_from = 15, intensity_to = 300))
## Only mass peaks with intensities between the specified limits are
## retained
intensity(res)
## Compared to the original intensities
intensity(sps)

## Note that the spectra variable `"name"` was lost during conversion of
## the MS data between R and Python:
sps$name
any(spectraVariables(res) == "name")

## Only spectra variables defined by `spectraVariableMapping()` are
## converted and thus retained:
spectraVariableMapping()

## We can also pass a custom *spectra variable mapping* with the `mapping`
## parameter to the `filterSpectriPy()` function. Below we create such
## a mapping by adding the translation of a spectra variable `"name"` to
## a metadata name `"compound_name"` to the default spectra variable
## mapping `defaultSpectraVariableMapping()`.
map <- c(defaultSpectraVariableMapping(), name = "compound_name")
map

## Repeat the filtering operation passing this mapping information:
res <- filterSpectriPy(
    sps, select_by_intensity(intensity_from = 15, intensity_to = 300),
    mapping = map)
res$name
```

MsBackendPy *A MS data backend for MS data stored in Python*

Description

The MsBackendPy allows to access MS data stored as matchms.Spectrum or spectrum_utils.spectrum.MsmsSpectrum objects from the *matchms* respectively *spectrum_utils* Python library directly from R. The MS data (peaks data or spectra variables) are translated on-the-fly when accessed. Thus, the MsBackendPy allows a seamless integration of Python MS data structures into Spectra::Spectra() based analysis workflows.

The MsBackendPy object is considered *read-only*, i.e. it does not provide functionality to replace the peaks data from R. However, it is possible to directly change the data in the referenced Python variable.

Usage

```
## S4 method for signature 'MsBackendPy'
backendInitialize(
  object,
  pythonVariableName = character(),
  spectraVariableMapping = defaultSpectraVariableMapping(),
  pythonLibrary = c("matchms", "spectrum_utils"),
  ...,
  data
)

## S4 method for signature 'MsBackendPy'
length(x)

## S4 method for signature 'MsBackendPy'
spectraVariables(object)

## S4 method for signature 'MsBackendPy'
spectraData(object, columns = spectraVariables(object), drop = FALSE)

## S4 method for signature 'MsBackendPy'
peaksData(object, columns = c("mz", "intensity"), drop = FALSE)

## S4 method for signature 'MsBackendPy'
x$name

## S4 replacement method for signature 'MsBackendPy'
spectraVariableMapping(object) <- value

## S4 replacement method for signature 'Spectra'
spectraVariableMapping(object) <- value

## S4 method for signature 'MsBackendPy'
spectraVariableMapping(object, value)

reindex(object)
```

Arguments

object	A MsBackendPy object.

pythonVariableName
: For backendInitialize(): character(1) with the name of the variable/Python attribute that contains the list of matchms.Spectrum objects with the MS data.

spectraVariableMapping
: For backendInitialize(): named character with the mapping between spectra variable names and (matchms.Spectrum) metadata names. See defaultSpectraVariableMappin for more information and details.

pythonLibrary
: For backendInitialize(): character(1) specifying the Python library used to represent the MS data in Python. Can be either pythonLibrary = "matchms" (the default) or pythonLibrary = "spectrum_utils".

...
: Additional parameters.

data
: For backendInitialize(): DataFrame with the full MS data (peaks data and spectra data). Currently not supported.

x
: A MsBackendPy object

columns
: For spectraData(): character with the names of columns (spectra variables) to retrieve. Defaults to spectraVariables(object). For peaksData(): character with the names of the peaks variables to retrieve.

drop
: For spectraData() and peaksData(): logical(1) whether, when a single column is requested, the data should be returned as a vector instead of a data.frame or matrix.

name
: For $: character(1) with the name of the variable to retrieve.

value
: Replacement value(s).

Details

The MsBackendPy keeps only a reference to the MS data in Python (i.e. the name of the variable in Python) as well as an index pointing to the individual spectra in Python but no other data. Any data requested from the MsBackendPy is accessed and translated on-the-fly from the Python variable. The MsBackendPy is thus an interface to the MS data, but not a data container. All changes to the MS data in the Python variable (performed e.g. in Python) immediately affect any MsBackendPy instances pointing to this variable.

Special care must be taken if the MS data structure in Python is subset or its order is changed (e.g. by another process). In that case it might be needed to re-index the backend using the reindex() function: object <- reindex(object). This will update (replace) the index to the individual spectra in Python which is stored within the backend.

Value

See description of individual functions for their return values.

MsBackendPy methods

The MsBackendPy supports all methods defined by the Spectra::MsBackend() interface for access to MS data. Details on the invidual functions can also be found in the main documentation in the *Spectra* package (i.e. for Spectra::MsBackend()). Here we provide information for functions with specific properties of the backend.

- backendInitialize(): initializes the backend with information from the referenced Python variable (attribute). The name of this attribute, ideally stored in the associated Python session, is expected to be provided with the pythonVariableName parameter. The optional spectraVariableMapping parameter allows to provide additional, or alternative, mapping of

Spectra's *spectra variables* to metadata in the matchms.Spectrum objects. See defaultSpectraVariableMappin (the default) for more information. Parameter pythonLibrary must be used to specify the Python library representing the MS data in Python. It can be either pythonLibrary = "matchms" (the default) or pythonLibrary = "spectrum_utils". The function returns an initialized instance of MsBackendPy.

- peaksData(): extracts the peaks data matrices from the backend. Python code is applied to the data structure in Python to extract the *m/z* and intensity values as a list of (numpy) arrays. These are then translated into an R list of two-column numeric matrices. Because Python does not allow to name columns of an array, an additional loop in R is required to set the column names to "mz" and "intensity".
- spectraData(): extracts the spectra data from the backend. Which spectra variables are translated and retrieved from the Python objects depends on the backend's spectraVariableMapping(). All metadata names defined are retrieved and added to the returned DataFrame (with eventually missing *core* spectra variables filled with NA).
- spectraVariables(): retrieves available spectra variables, which include the names of all metadata attributes in the matchms.Spectrum objects and the *core* spectra variables Spectra::coreSpectraVariab
- spectraVariableMapping(): get the currently defined mapping for spectraVariables() of the backend.
- spectraVariableMapping<-: replaces the spectraVariableMapping of the backend (see setSpectraVariableMapping() for details and description of the expected format).

Additional helper and utility functions

- reindex(): update the internal *index* to match 1:length(object). This function is useful if the original data referenced by the backend was subset or re-ordered by a different process (or a function in Python).

Note

As mentioned in the *details* section the MS data is completely stored in Python and the backend only references to this data through the name of the variable in Python. Thus, each time MS data is requested from the backend, it is retrieved in its **current** state. If for example data was transformed or metadata added or removed in the Python object, it immediately affects the Spectra/backend.

Author(s)

Johannes Rainer and the EuBIC hackathon team

Examples

```
## Loading an example MGF file provided by the SpectriPy package.
## As an alternative, the data could also be imported directly in Python
## using:
## import matchms
## from matchms.importing import load_from_mgf
## s_p = list(load_from_mgf(r.fl))
library(Spectra)
library(SpectriPy)
library(MsBackendMgf)

fl <- system.file("extdata", "mgf", "test.mgf", package = "SpectriPy")
s <- Spectra(fl, source = MsBackendMgf())
s
```

```
## Translating the MS data to Python and assigning it to a variable
## named "s_p" in the (*reticulate*'s) `py` Python environment. Assigning
## the variable to the Python environment has performance advantages, as
## any Python code applied to the MS data does not require any data
## conversions.
py_set_attr(py, "s_p", rspec_to_pyspec(s))


## Create a `MsBackendPy` representing an interface to the data in the
## "s_p" variable in Python:
be <- backendInitialize(MsBackendPy(), "s_p")
be

## Create a Spectra object which this backend:
s_2 <- Spectra(be)
s_2

## Available spectra variables: these include, next to the *core* spectra
## variables, also the names of all metadata stored in the `matchms.Spectrum`
## objects.
spectraVariables(s_2)

## Get the full peaks data:
peaksData(s_2)

## Get the peaks from the first spectrum
peaksData(s_2)[[1L]]

## Get the full spectra data:
spectraData(s_2)

## Get the m/z values
mz(s_2)

## Plot the first spectrum
plotSpectra(s_2[1L])


########
## Using the spectrum_utils Python library

## Below we convert the data to a list of `MsmsSpectrum` object from the
## spectrum_utils library.
py_set_attr(py, "su_p", rspec_to_pyspec(s,
    spectraVariableMapping("spectrum_utils"), "spectrum_utils"))

## Create a MsBackendPy representing this data. Importantly, we need to
## specify the Python library using the `pythonLibrary` parameter and
## ideally also set the `spectraVariableMapping` to the one specific for
## that library.
be <- backendInitialize(MsBackendPy(), "su_p",
    spectraVariableMapping = spectraVariableMapping("spectrum_utils"),
    pythonLibrary = "spectrum_utils")
be

## Get the peaks data for the first 3 spectra
```

```
peaksData(be[1:3])

## Get the full spectraData
spectraData(be)

## Extract the precursor m/z
be$precursorMz
```

Index