Package 'ProtGenerics'

October 24, 2025

Title Generic infrastructure for Bioconductor mass spectrometry packages

Description S4 generic functions and classes needed by Bioconductor proteomics packages.

Version 1.41.0

Author Laurent Gatto <laurent.gatto@uclouvain.be>, Johannes Rainer <johannes.rainer@eurac.edu>

Maintainer Laurent Gatto <laurent.gatto@uclouvain.be>

biocViews Infrastructure, Proteomics, MassSpectrometry

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

Depends methods

Suggests testthat

License Artistic-2.0

NeedsCompilation no

RoxygenNote 7.3.2

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

git_branch devel

git_last_commit a9662c5

git_last_commit_date 2025-04-15

Repository Bioconductor 3.23

Date/Publication 2025-10-24

Contents

`backendInitialize` *General backend methods*

Description

These methods are used for implementations of *backends* e.g. for `Spectra` or `Chromatograms` object to initialize the backend, merge backends or extract specific information from them. See the respective help pages (e.g. in the `Spectra` or `Chromatograms` packages) for information on the actual implementations of these methods.

Usage

```
backendInitialize(object, ...)

isReadOnly(object)

setBackend(object, backend, ...)

backendMerge(object, ...)

backendBpparam(object, ...)

backendParallelFactor(object, ...)

supportsSetBackend(object, ...)
```

Arguments

`object`	The *backend* object.
`...`	Optional parameters.
`backend`	A *backend* object.

extractByIndex *Extracting elements by index*

Description

The `extractByIndex()` method allows to subset an object (or extract elements from it) by providing their `integer` indices.

Usage

```
extractByIndex(object, i)
```

Arguments

`object`	The object to subset/from which to extract elements.
`i`	`integer` with the indices.

filterFeatures *Filter features*

Description

Implementations of this generic filter function are supposed to filter *features* in `object` based on a filter criteria defined by parameter `filter`.

Usage

```
filterFeatures(object, filter, ...)
```

Arguments

`object`	The object to filter.
`filter`	The filtering criteria on which `object` should be filtered.
`...`	Optional parameters.

filterSpectra *Filter Spectra*

Description

Implementations of this generic filter function are supposed to filter *spectra* (e.g. within a `Spectra` object) based on filter criteria defined with parameter `filter`.

Usage

```
filterSpectra(object, filter, ...)
```

Arguments

`object`	The object to filter.
`filter`	The filtering criteria based on which `object` should be filtered.
`...`	Optional parameters.

Param *Generic parameter class*

Description

The ‘Param‘ class is a virtual class which can be used as *base* class from which *parameter* classes can inherit.

The methods implemented for the ‘Param‘ class are:

- ‘as.list‘: coerces the ‘Param‘ class to a ‘list‘ with list elements representing the object’s slot values, names the slot names. *Hidden* slots (i.e. those with a name starting with ‘.‘) are not returned. In addition, a ‘Param‘ class can be coerced to a ‘list‘ using ‘as(object, "list")‘.

- ‘show‘: prints the content of the ‘Param‘ object (i.e. the individual slots and their value).

Usage

```
## S4 method for signature 'Param'
as.list(x, ...)

## S4 method for signature 'Param'
show(object)
```

Arguments

`x`	‘Param‘ object.
`...`	ignored.
`object`	‘Param‘ object.

Author(s)

Johannes Rainer

`peaksData` *Get or set MS peak data*

Description

These methods get or set mass spectrometry (MS) peaks data, which can be m/z, intensity or retention time values. See the respective help pages (e.g. in the `Spectra` or `Chromatograms` packages) for information on the actual implementations of these methods.

Usage

```
peaksData(object, ...)

peaksData(object) <- value

peaksVariables(object, ...)
```

Arguments

`object`	A data object.
`...`	Optional parameters.
`value`	Replacement for peaks data.

`processingQueue` *Processing Queue*

Description

These methods are related to the *processing queue* implemented in the [Spectra](https://github.com/RforMassSpectrometry and [Chromatograms](https://github.com/RforMassSpectrometry/Chromatograms) packages.

- ‘addProcessing()‘ adds a processing step to the processing queue.

- ‘applyProcessing()‘ execute the processing queue replacing the original data in ‘object‘ with the processed one.

- ‘processingChunkSize()‘ and ‘processingChunkSize()<-‘ are supposed to get and set the number of elements (e.g. spectra) for which the data is loaded into memory and processed at a time.

- ‘processingChunkFactor()‘: defines a ‘factor‘ that can be used to split ‘object‘ into chunks defined by the length of ‘object‘ and its ‘processingChunkSize()‘.

Usage

```
processingChunkSize(object, ...)

processingChunkSize(object, ...) <- value

addProcessing(object, ...)

applyProcessing(object, ...)

processingChunkFactor(object, ...)
```

Arguments

object	The object with the processing queue.
...	Additional parameters to be defined.
value	The replacement value.

`ProcessingStep` *Processing step*

Description

Class containing the function and arguments to be applied in a lazy-execution framework.

Objects of this class are created using the `ProcessingStep()` function. The processing step is executed with the `executeProcessingStep()` function.

Usage

```
ProcessingStep(FUN = character(), ARGS = list())

## S4 method for signature 'ProcessingStep'
show(object)

executeProcessingStep(object, ...)
```

Arguments

FUN	`function` or `character` representing a function name.
ARGS	`list` of arguments to be passed along to `FUN`.
object	`ProcessingStep` object.
...	optional additional arguments to be passed along.

Details

This object contains all relevant information of a data analysis processing step, i.e. the function and all of its arguments to be applied to the data. This object is mainly used to record possible processing steps of a `Spectra` or `OnDiskMSnExp` object (from the `Spectra` and `MSnbase` packages, respectively).

Value

The `ProcessingStep` function returns and object of type `ProcessingStep`.

Author(s)

Johannes Rainer

Examples

```
## Create a simple processing step object
ps <- ProcessingStep(sum)

executeProcessingStep(ps, 1:10)
```

`ProtGenerics` *S4 generic functions for Bioconductor proteomics infrastructure*

Description

These generic functions provide basic interfaces to operations on and data access to proteomics and mass spectrometry infrastructure in the Bioconductor project.

For the details, please consult the respective methods' manual pages.

Usage

```
psms(object, ...)
peaks(object, ...)
modifications(object, ...)
database(object, ...)
rtime(object, ...)
tic(object, ...)
spectra(object, ...)
intensity(object, ...)
mz(object, ...)
peptides(object, ...)
proteins(object, ...)
accessions(object, ...)
scans(object, ...)
mass(object, ...)
```

```
ions(object, ...)
chromatograms(object, ...)
chromatogram(object, ...)
isCentroided(object, ...)
writeMSData(object, file, ...)
## and many more
```

Arguments

`object`	Object of class for which methods are defined.
`file`	for `writeMSData`

: the name of the file to which the data should be exported.

`...`	Further arguments, possibly used by downstream methods.

Details

When should one define a generics?:

Generics are appropriate for functions that have *generic* names, i.e. names that occur in multiple circumstances, (with different input classes, most often defined in different packages) or, when (multiple) dispatching is better handled by the generics mechanism rather than the developer. The dispatching mechanism will then automatically call the appropriate method and save the user from explicitly calling `package::method` or the developer to handle the multiple input types cases. When no such conflict exists or is unlikely to happen (for example when the name of the function is specific to a package or domain, or for class slots accessors and replacement methods), the usage of a generic is arguably questionable, and in most of these cases, simple, straightforward functions would be perfectly valid.

When to define a generic in `ProtGenerics`?:

`ProtGenerics` is not meant to be the central package for generics, nor should it stop developers from defining the generics they need. It is a means to centralise generics that are defined in different packages (for example `mzR::psms` and `mzID::psms`, or `IRanges::score` and `mzR::score`, now `BioGenerics::score`) or generics that live in a rather big package (say `mzR`) on which one wouldn't want to depend just for the sake of that generics' definition.

The goal of `ProtGenerics` is to step in when namespace conflicts arise so as to to facilitate inter-operability of packages. In case such conflict appears due to multiple generics, we would (1) add these same definitions in `ProtGenerics`, (2) remove the definitions from the packages they stem from, which then (3) only need to import `ProtGenerics`. This would be very minor/straightforward changes for the developers and would resolve issues when they arise.

More generics can be added on request by opening an issue or sending a pull request on:

https://github.com/lgatto/ProtGenerics

Author(s)

Laurent Gatto

See Also

- The **BiocGenerics** package for S4 generic functions needed by many Bioconductor packages.
- showMethods for displaying a summary of the methods defined for a given generic function.
- selectMethod for getting the definition of a specific method.
- setGeneric and setMethod for defining generics and methods.

Examples

```
## List all the symbols defined in this package:
ls('package:ProtGenerics')

## Not run:
    ## What methods exists for 'peaks'
    showMethods("peaks")

    ## To look at one method in particular
    getMethod("peaks", "mzRpwiz")

## End(Not run)
```

Index