Package 'ProtGenerics'

October 14, 2021

Title Generic infrastructure for Bioconductor mass spectrometry packages

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

Version 1.24.0

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/lgatto/ProtGenerics

Depends methods

Suggests testthat

License Artistic-2.0

NeedsCompilation no

RoxygenNote 7.1.1

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

git_branch RELEASE_3_13

git_last_commit 50e7e66

git_last_commit_date 2021-05-19

Date/Publication 2021-10-14

R topics documented:

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

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, ...)
```

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