maspy Documentation

Release

Author

Jun 06, 2017

Contents

Contents:

CHAPTER 1

Introduction to basic concepts of MasPy

Contents:

MS spectra in MasPy

The mzML file format

Every vendor software produces mass spectrometer output files in a different proprietary format. It is a difficult and time consuming task for software developers to support all of these different formats and format versions. Therefore the file format mzML has been developed by the Proteomics Standards Initiative (PSI) as the community standard for representation of mass spectrometry results. mzML is an open, XML- based format that not only allows to store recorded mass spectrum information but also metadata of the instrument configuration, acquisition settings, software used for data processing and sample descriptions. Ultimately, it is desirable to universally use mzML for archiving, sharing, and processing of mass spectrometry data and thus for all software to support and use the mzML format.

Note: Refer to www.psidev.info for details on the XML schema definition and mzML file specifications, see also the publication Mass Spectrometer Output File Format mzML)

Note: We recommend using ProteoWizard for conversion of vendor format files to mzML. The software can be downloaded from their website, a detailed protocol how to use ProteoWizard can be found here.

The raw spectral data recorded by an instrument can be either stored as profile or centroid data. Meassured mass spectra are initially recorded in profile mode, where each mass peak is represented by a number of m/z and intensity values describing a peak shape. In centroid mode this information is reduced to the centroid of the peak shape, storing only one single pair of a dinstinct m/z value and an intensity. The process of converting profile data to centroid data is called peak picking and can be applied as a filter while converting vendor format files to mzML files using ProteoWizard, see the ProteoWizard protocol. The representation as centroid data is easier to work with, saves memory and is sufficient for most applications. Therefore we recommend the utilization of centroid data for MasPy.

MsrunContainer

Modern mass spectrometers can generate tens of thousands of spectra per hour resulting in huge mzML files. Opening and parsing such large XML files takes a lot of time. MzML files can contain a byte-offset index which allows directly reading certain spectra without parsing the whole file. This can increase performance when only one or a few specific spectra have to be accessed at a time.

The actual spectral information takes up to largest part of a typical mzML file. However, sometimes only a certain type of information needs to be accessed, for example the spectrum metadata. Therefore we split the information that is contained in mzML files into four data groups; run metadata (`Rm`), spectrum metadata items (`Smi`), spectrum array items (`Sai`) and chromatogram items (`Ci`). Each of these data groups is stored separately in MasPy and has its own file type, thus it can be accessed, saved and loaded independently of the others. All four data types are stored in the MasPy class *`MsrunContainer`*. Altough the data is split into multiple parts, all information originally contained in an mzML file is still present. This allows the conversion from MsrunContainer to mzML at any given time. #TODO: Why do we want to be able to export mzML files? (Preffered data format for archiving and sharing data and to use as input for other software packages)

See tutorial/docstrings xxx for details on the MsrunContainer file format. #TODO:

Fig.: MsrunContainer #TODO: make figure

- run metadata
- spectrum metadata items
- spectrum array items
- chromatogram items
- spectrum items

Run metadata (`Rm`)

The run metadata element contains all information of an mzML file, which is not directly part of the acquired spectra and chromatograms. This covers, amongst others, a description of the instrument configuration, a list of software used for data processing and a list of applied data processing steps. In addition it is possible to add contact information and a description of the analyzed samples to the mzML file. In MasPy all of these mzML elements are converted to an `lxml.etree.Element` and stored in `MsrunContainer.rmc` (Rm container).

Note: Software which is used to process data of an mzML file should be listed in the mzML element “softwareList”, and all applied data processing steps should be documented in the “dataProcessingList” element.

Spectrum array item (`Sai`), spectrum metadata item (`Smi`)

An mzML spectrum element contains all information of an acquired MS spectrum, including numerical arrays containing at least recorded m/z and intensity values of the observed ions but also plenty of metadata describing for example details of the acquisition like base peak m/z and intensity, scan start time, ms level, MS2 isolation window or precursor information of MS2 scans. In MasPy this information is split into a metadata containing part and the spectrum array data and put into two separate data structures; spectrum metadata item (`Smi`) and spectrum array item (`Sai`), respectively. `Smi` elements are stored in `MsrunContainer.smic` (Smi container) and `Sai` elements in `MsrunContainer.saic` (Sai container). In order to recreate an mzML spectrum element the information of both MasPy data types (`Smi` and `Sai`) is necessary.

Chromatogram item (`Ci`)

An mzML chromatogram element is similar to a spectrum element, containing metadata and numerical arrays. Common chromatogram types are `total ion current chromatogram`, `selected ion current chromatogram` and `basepeak chromatogram`. All of them contain time and intensity data points, however, other chromatogram types can also contain absorption or emission values instead of intensities. In the current MasPy implementation chromatogram elements are not split into two data types but the metadata and array information is put into one single data structure called chromatogram item (`Ci`), which is stored in `MsrunContainer.cic` (Ci container).

Spectrum item (`Si`)

The mzML file serves as a data container for active data processing but also for data sharing and archiving. Thus the spectrum elements contain a lot of metadata information not needed for most data analysis applications. In addition all information stored in spectrum elements have to be in accordance with the mzML xml scheme definition and the Controlled Vocabularies (CV's) of the PSI, see. Altough in principle this standardization is beneficial and perfectly reasonable, when actively working with the data it is not always required and can make things unnecessarily complicated.

To circumvent this problem MasPy provides a simpler data type for working with spectrum metadata, called spectrum item (*`Si`*). The `Si` class has a flat structure, meaning that attributes are not nested inside other elements but are stored directly as attributes of the class. `Si` attributes can be manipulated without restrictions and new attributes can simply be added. Specific functions can be used to selectively extract information from `Smi`. This allows import only the currently needed spectrum metadata attributes, like retention time, ms level or MS2 precursor information, thereby making the `Si` more memory efficient. In order to make lasting changes to the mzML file `Si` attributes have to be translated to the respective `Smi` elements. These changes however have to strictly follow the mzML specifications and syntax. Thus it is recommend to use existing functions or implement new ones that make changes to `Smi` elements in a controlled manner.

Each spectrum present in an mzML file is therefore represented threefold in MasPy. First the `Smi` contains a complete representation of all metadata information present in an mzML spectrum element. However, this data type is not intended to be used for standard data analysis and will normally only be accessed to make lasting, documented changes to spectrum metadata and for generating new mzML files. Second the `Sai` contains the actual ion information recorded by the mass spectrometer. This data type will be used whenever the ion spectra have to be analyzed or manipulated. In addition it is also required for generating new mzML files. And third the `Si`, which can be considered as the spectrum metadata workspace in MasPy, allowing convenient access to metadata and simple processing of this data without directly altering the original mzML information. This data type will be used for most data processing and analysis steps in MasPy.

MsrunContainer.info

MsrunContainer.info -> which specfiles are present, what is the current path (used for loading or saving) , which data types are currently imported

MasPy file formats

This section will contain information about how the data contained in an MsrunContainer is written to the hard drive. (one file type per data type: mrc_rm, mrc_si, mrc_sai, mrc_smi, mrc_ci)

Basic code examples

Importing an mzML file

mzML files can be imported by using the function *maspy.reader.importMzml()*, the imported specfile is then added to the `MsrunContainer` instance passed to the function.

```
import maspy.core
import maspy.reader

mzmlfilepath = 'filedirectory/specfile_name_1.mzML'
msrunContainer = maspy.core.MsrunContainer()
maspy.reader.importMzml(mzmlfilepath, msrunContainer)
```

Saving an MsrunContainer to the hard disk

An `MsrunContainer` can be saved to the hard disk by calling its *.save()* method.

```
msrunContainer.save()
```

By default all files are saved into the folder specified in `.info`. This can be altered by changing the `path` variable in `.info` or temporarely by passing the "path" parameter to `.save()`.

```
msrunContainer.save(path='../an_alternative_location')
```

In addition, multiple parameters can be set to specify which part of the data should be written to the hard disk. The keywords "rm", "ci", "smi", "sai" and "si" can be set to `True` or `False` and specify which container types are selected for saving. By default all of them are set to `False` which is however interpreted as selecting all of them. Setting at least one to `True` changes this behaviour and only the specified ones are selected. If multiple specfiles are present in an `MsrunContainer` it is possible to only select a subset for saving by passing the "specfiles" argument to `.save()`. The value of "specfiles" can either be the name of one single specfile or a list of specfile names. In the following example only the spectrum array item container (saic) and the spectrum metadata item container (smic) of the specfiles "specfile_name_1" and "specfile_name_3" are saved.

```
msrunContainer.save(specfiles=["specfile_name_1", "specfile_name_3"],
                    sai=True, smi=True
                    )
```

Loading an MsrunContainer from the hard disk

Before loading an `MsrunContainer` from the hard disk, a specfile entry has to be added to its `.info` attribute. This can be done by calling *.addSpecfile()* with the name of the specfile and the path to the filedirectory. Afterwards the files can be loaded by calling *.load()*, which will import all specfiles present in `.info` and update the `status` variable of `.info`.

```
>>> msrunContainer = maspy.core.MsrunContainer()
>>> msrunContainer.addSpecfile('specfile_name_1', 'filedirectory')
>>> msrunContainer.info
{u'specfile_name_1': {u'path': u'filedirectory',
                      u'status': {u'ci': False,
                                  u'rm': False,
                                  u'sai': False,
                                  u'si': False,
                                  u'smi': False}}}
```

```
>>> msrunContainer.load()
>>> msrunContainer.info
{u'specfile_name_1': {u'path': u'filedirectory',
                      u'status': {u'ci': True,
                                  u'rm': True,
                                  u'sai': True,
                                  u'si': True,
                                  u'smi': True}}}
```

Similar to saving only parts of an `MsrunContainer` it is also possible to only select a subset of specfiles present in `.info` and specify which data types are imported.

```
>>> msrunContainer = maspy.core.MsrunContainer()
>>> msrunContainer.addSpecfile('specfile_name_1', 'filedirectory')
>>> msrunContainer.info
{u'specfile_name_1': {u'path': u'filedirectory',
                      u'status': {u'ci': False,
                                  u'rm': False,
                                  u'sai': False,
                                  u'si': False,
                                  u'smi': False}}}
>>> msrunContainer.load(specfiles='specfile_name_1', sai=True, smi=True)
>>> msrunContainer.info
{u'specfile_name_1': {u'path': u'filedirectory',
                      u'status': {u'ci': False,
                                  u'rm': False,
                                  u'sai': True,
                                  u'si': False,
                                  u'smi': True}}}
```

Deleting data from an MsrunContainer

If specific data types are not needed anymore, they can be removed to free memory. This can be done by using *`.removeData()`* and parsing arguments to specify specfiles and which data types to remove. It is recommended to always use this method to remove data instead of manually deleting container entries, because using `.removeData` automatically updates the `.info` attribute of the `MsrunContainer`. The following command removes the `Sai` and `Smi` items of the specfile "specfile_name_1".

```
>>> msrunContainer.info
{u'specfile_name_1': {u'path': u'filedirectory',
                      u'status': {u'ci': True,
                                  u'rm': True,
                                  u'sai': True,
                                  u'si': True,
                                  u'smi': True}}}
>>> msrunContainer.removeData('specfile_name_1', sai=True, smi=True)
>>> msrunContainer.info
{u'specfile_name_1': {u'path': u'filedirectory',
                      u'status': {u'ci': True,
                                  u'rm': True,
                                  u'sai': False,
                                  u'si': True,
                                  u'smi': False}}}
```

A specfile can be completely removed from an `MsrunContainer` by calling *`.removeSpecfile()`*, which deletes all data from the containers and in addition the entry from the `.info` attribute.

```
msrunContainer.removeSpecfile('specfile_name_1')
```

Exporting specfiles from MsrunContainer to mzML files

After working in MasPy it might be desirable to export the MsrunContainer back into an mzML file which can be used as input for another software or simply for archiving and sharing mass spectrometry data. An mzML file is generated by using the function *maspy.writer.writeMzml()* and passing at least the `specfile` name that should be exported, an `MsrunContainer` and the `output directory`. In order to write a valid and complete mzML file all data types except for `Si` have to be present in the `MsrunContainer`.

```
import maspy.writer
maspy.writer.writeMzml('specfile_name_1', msrunContainer, '/filedirectory')
```

Note: Optionally it is possible to supply a list of `spectrumIds` and `chromatogramIds` to only select a subset of spectra and chromatograms that should be written to the mzML file. The supplied lists of element ids have to be sorted in the order they should be written to the mzML file.

Accessing data from an MsrunContainer

#TODO: *examples of .getItem, .getArrays, ... *

Spectrum identifications in MasPy

In bottom up proteomic experiments proteins are analysed indirectly by peptides generated by proteolytic digestion. In most cases the resulting peptides are separated by liquid chromatography before they are ionized and subsequently analysed by the mass spectrometer in a setup called liquid chromatography-tandem mass spectrometry (LC-MS/MS). During the chromatographic separation the mass spectrometer records in certain intervals the mass to charge ratios (m/z) of all ions eluting at a given time point, which produces so called MS1 spectra. However, altough the m/z value of a peptide ion is known with very high accuracy, it is not possible to infer its amino acid sequence. This is because a huge number of theoretical peptides exist which have nearly or exactly the same mass but a different amino acid sequence. To solve this, ions present in the MS1 scan are isolated consequtively by a mass filter, fragmented and the m/z values of the resulting fragment ions are recorded, which generates an MS2 spectrum. It is also possible to isolate a fragment ion from an MS2 spectrum, fragment it again and measure the resulting ions, which is then called MS3. This procedure could be repeated multiple times and would result in consequent MSn scans, where "n" stands for the number of isolation and fragmentation cycles +1. Peptide sequence identification can now be inferred by comparing the observed MSn spectra with theoretical fragmentation spectra generated in silico. This procedure is called peptide spectrum matching. Another possibility is to compare the observed MS2 spectra with already identified observed fragmentation spectra from a library, which is called spectral library matching. For details see Protein Analysis by Shotgun/Bottom-up Proteomics #TODO: open access would be better, is it open?

The mzIdentML file format

#TODO: description of mzIdentML format

Representation of spectrum identifications in MasPy

Elements describing the identification of a peptide from a fragmentation spectrum are called spectrum identification items (*Sii*) in MasPy. The term `Sii` is interchangeable with the more commonly used term peptide spectrum match (PSM). However, we choose `Sii` since it is in analogy to the mzIdentML format which we are planning to support in the future.

The `Sii` class has a very simple structure with all its variables being directly stored as attributes of the class. The attributes `id` and `specfile` are mandatory and used to unambiguously link the `Sii` to a `Si` element of an `MsrunContainer`. The `Sii.id` is typically the scan number of a spectrum and should be equal to the `Si.id` entry, the `Si.specfile` refers to the specfile name which is used to identify a single ms-run. Other attributes can be manipulated without restrictions and new attributes can simply be added.

Further attributes which can be necessary for a reasonable utilization of `Sii` and their naming convention in MasPy:

- `peptide` the peptide sequence containing amino acid modifications in the MasPy format, see below.
- `sequence` the plain amino acid sequence of the spectrum identification, does not contain modifications.
- `score` or any other score attribute name which is used to rank the quality of a spectrum identifications. The name of this attribute and wheter a large or a small number indicates a higher confidence is specified in *SiiContainer.info*.
- `isValid` can be used to flag if a Sii has passed a given quality threshold or has been validated as correct.
- `rank` the rank of this Sii compared to others for the same MSn spectrum. The rank is based on the specified score attribute.
- `charge` the charge state of the identified precursor ion.
- `rt` the retention time in seconds of the corresponding spectrum.
- `obsMz` the experimentally observed mass to charge ratio of the precursor ion (Dalton / charge). Usually the monoisotopic ion.
- `obsMh` the experimentally observed mass to charge ratio of the precursor ion, calculated for the mono protonated ion (Dalton / charge). Usually the monoisotopic ion.
- `obsMass` the experimentally observed not protonated mass of precursor ion, calculated by using the mz and charge values (Dalton / charge). Usually the monoisotopic mass.
- `excMz` the exact calculated mass to charge ratio of the peptide (Dalton / charge). Usually the monoisotopic ion.
- `excMh` the exact calculated mass to charge ratio of the peptide, calculated for the mono protonated peptide (Dalton / charge). Usually the monoisotopic ion.
- `excMass` the exact calculated mass of the not protonated peptide (Dalton / charge). Usually the monoisotopic mass.

Note: The amino acid sequence itself is commonly written in single letter code. However, there is no common style how to depict amino acid modifications in a linear string. Very often each modification is represented by a single symbol or a short string specifically highlighted for example by using brackets. This modification represenation is then written next to the modified amino acid residue, i.e. on the right side.

In MasPy we decided to highlight modifications by using square brackets positioned right of the modified amino acid. It is possible to add multiple modifications to one single residue by writing multiple bracket pairs, eg `PEP[mod1][mod2]TIDE`. This format allows simple parsing of peptide strings to retrieve modifications and their position in the amino acid sequence. In addition every character, except square brackets, could be used as a symbol for an additional amino acid.

Unimod provides a comprehensive database of protein modifications and is to our knowledge widely-used in the field of mass spectrometry based proteomics. Therefore we decided to refer to the unimod accession number whenever a modification is present in the database. Such modifications are then written in the form of `[u:X]`, where X is the unimod accession number. Modifications not present in the database should be represented by a short acronym, for example `[DSS]`. Such additional modifications have to be added to the MasPy modification database. (at the moment this is only a dictionary *`maspy.constants.aaModComp`*)

The spectrum identification item container (SiiContainer)

The *`SiiContainer`* is used to store spectrum identification results of one or multiple specfiles. The container allows saving and loading of imported results and provides methods for convenient access to the data.

Importing peptide spectrum matching results

During the import all PSMs have to be converted to `Sii` and added to `SiiContainer.container`. Since for one spectrum multiple `Sii` can exist, they have to be ranked according to how well they can explain the observed fragmentation ions, typically described by a score or the q-value. All `Sii` of the same spectrum are put into a list, ranked and sorted according to a user defined score. This sorted list is then stored in `.container`, for details see below. Only the `Sii` at the first position of this list gets the attribute `.isValid` set to `True`, this is even the case if multiple `Sii` with `.rank == 1` exist. Afterwards all valid `Sii` are additional evaluated if they surpass a user defined quality threshold, typically this threshold is a false discovery rate (FDR) of 1%.

The import routines currently provided by MasPy are not very extensive, covering only the import from percolator .tsv files of certain PSM search engines and mzIdentML files generated by MS-GF+. However, adjusting the existing methods to any .tsv file should be possible within minutes and will be added on demand. Two things are important to consider when doing this:

- The spectrum identifier (scan number) is not always present in a separate field. It is very often part of a so called scan header string, which also contains the specfilename and sometimtes the precursor charge state. Thus it is necessary to provide a function which extracts the scan number.
- For the import it is necessary to provide a function which translates a modification containing peptide string into the aforementioned MasPy representation. In most cases this can be achieved by a simple mapping function:

```
>>> def translatePeptide(peptide, modificationMapping):
>>>     for oldMod, maspyMod in modificationMapping:
>>>         peptide = peptide.replace(oldMod, '[' + maspyMod + ']')
>>>     return peptide

>>> modificationMapping = [('#', 'u:21'), ('*', 'u:35')]
>>> translatePeptide('S#PEPM*K', modificationMapping)
u'S[u:21]PEPM[u:35]K'
```

Note: It might be necessary for the function that translates a modified peptide string into the MasPy format to be able to deal with modification strings which are a substring of another modification string, for example "ox" and "diox" in PoxEPdioxTIDE. In such a case if "ox" is simply converted both instances would be affected and the "di" would remain untreated. For most cases this can be solved by replacing the modifications which are a substring of another modification after the others.

The minimal information that should be imported from peptide spectrum matching results are the **scan identifier**, **modified peptide sequence** and a **score**, which can be used to apply a quality cut off. Other parameters can either be generated from the modified peptide sequence (calculated mass, plain amino acid sequence) or transferred from the `MsrunContainer`

Basic code examples

Importing spectrum identification results

A percolator tab separated file can be imported by using the function *maspy.reader.importPercolatorResults()*, the imported `Sii` elements are then added to the `SiiContainer` instance passed to the function.

```
import maspy.core
import maspy.reader

siiContainer = maspy.core.SiiContainer()
maspy.reader.importPercolatorResults(siiContainer, 'filelocation/out.tsv',
                                     'specfile_name_1', 'psmEngine')
```

If necessary, spectrum attributes can be added from the `MsrunContainer` by using the function *SiiContainer.addSiInfo*. This adds the selected attributes to all `Sii` elements of the specified specfiles.

```
import maspy.core
import maspy.reader

mzmlfilepath = 'filedirectory/specfile_name_1.mzML'
msrunContainer = maspy.core.MsrunContainer()
maspy.reader.importMzml(mzmlfilepath, msrunContainer)

siiContainer.addSiInfo(msrunContainer, specfiles='specfile_name_1',
                       attributes=['obsMz', 'rt', 'charge']
                       )
```

It is also possible to calculate the exact mass for all `Sii` elements of the specified specfiles by using the function *SiiContainer.calcMz()*. The calculated mass to charge ratio is written to the attribute `.excMz`.

```
siiContainer.calcMz(specfiles='specfile_name_1')
```

Accessing data stored in a SiiContainer

There are multiple ways how to access single `Sii` elements stored in a `SiiContainer`. The method `SiiContainer.getValidItem()` can be used to directly access `Sii` which `.isValid` argument is `True` by using its specfile and identifier. If no such `Sii` exists for the specified identifier `None` is returned. In this example there is a valid entry for the identifier '10', but not for '11'.:

```
>>> sii = siiContainer.getValidItem('specfile_name_1', '10')
>>> sii.isValid
True
>>> sii == None
False
>>> sii = siiContainer.getValidItem('specfile_name_1', '11')
>>> sii == None
True
```

It is possible to access all `Sii` elements of a given identifier by directly accessing the container *SiiContainer.container*. In this example, there are multiple `Sii` elements present for the same spectrum, but only one is valid.

```
>>> siiContainer.container['specfile_name_1']['10']
[<maspy.core.Sii at 0xb354a90>,
 <maspy.core.Sii at 0xb354c50>,
 <maspy.core.Sii at 0xb354b38>]
>>> for sii in siiContainer.container['specfile_name_1']['10']:
```

```
>>>     print(sii.id, sii.rank, sii.isValid)
10 1 True
10 2 False
10 3 False
```

By using the function `SiiContainer.getItems()` it is possible to iterate over all `Sii` elements present in the `SiiContainer`. Multiple arguments can be passed to the function that allow selecting only a specific subset of items but also to return the items in a sorted order. For details consult the docstring.

Depricated - work in progress

Outline

- Typical MS/MS experimental setup: MS1 detection -> isolation -> fragmentation
- Identification of MS/MS spectra -> assigning a peptide which generated the fragmentation spectrum

  This process can be described as peptide spectrum matching and is done by three major methods:

  - Comparison of theoretical spectra derived by an in silico digestion of a protein database
  - Comparison of observed spectra with spectra of known peptide origin (spectral library search)
  - De novo sequencing by comparing the mass differences of the observed ions to the actual mass differences of the peptide bulding blocks, amino acids and modified amino acids.
- How well does the assigned peptide explain the spectrum, described by an arbitrary score or a probability that the match is wrong
- What is the official format to record this identification information, mzIdentML. Not yet supported in MasPy
- Represenation of PSMs in MasPy (How to depict amino acid modifications in maspy)
- How are Sii stored in MasPy (SiiContainer), how to access Sii
- Importing PSM results into Maspy - minimal requirements (peptide, scanId, score), a function to translate modifications.

From thermo homepage

Tandem mass spectrometry (MS/MS) offers additional information about specific ions. In this approach, distinct ions of interest are selected based on their m/z from the first round of MS and are fragmented by a number of methods of dissociation. One such method involves colliding the ions with a stream of inert gas, which is known as collision-induced dissociation (CID) or higher energy collision dissociation (HCD). Other methods of ion fragmentation include electron-transfer dissociation (ETD) and electron-capture dissociation (ECD).

From abc and xyz (link)

Having determined the m/z values and the intensities of all the peaks in the spectrum, the mass spectrometer then proceeds to obtain primary structure (sequence) information about these peptides. This is called tandem MS, because it couples two stages of MS. In tandem MS, a particular peptide ion is isolated, energy is imparted by collisions with an inert gas (such as nitrogen molecules, or argon or helium atoms), and this energy causes the peptide to break apart. A mass spectrum of the resulting fragments — the tandem MS (also called MS/MS or MS2) spectrum — is then generated (Fig. 3c). In MS jargon, the species that is fragmented is called the 'precursor ion' and the ions in the tandem-MS spectrum are called 'product ions' (more endearingly, but less politically correct, they used to be described as parent and daughter ions). Note that the MS2 spectrum is the result of an ensemble of one particular precursor ion fragmenting at different amide bonds. Throughout the chromatographic run, the instrument will cycle through a sequence that consists of obtaining a mass spectrum followed by obtaining tandem mass spectra of the most abundant peaks that were found in this spectrum.

Peptide LC-MS features in MasPy

Chromatographic separation coupled directly to the mass spectrometer results in analytes to appear over time as they emerge from the chromatography column in a more or less Gaussian peak shape. The elution profile of an analyte can be recapitulated by using the spectral information present in the MS1 scans. The simplest way to do this is by extracting the intensities of an ion species with a given m/z value in consecutive MS1 scans as long as the ion is detectable. Combining the extracted intensities with the respective MS1 retention times results in a so called extracted ion chromatogram (EIC or XIC). The intensity area obtained by integrating the XIC is frequently used as a measure of abundance in label free quantification (LFQ) and stable isotope labeling (SIL) workflows.

In MS spectra each peptide species consists of an isotope envelope of multiple ion species with different m/z values. Combining the XICs of the different isotope states of the same analyte allows the inference of its charge state and therefore its mass. In addition the availability of more information results in more accurate intensity area estimates and thus increased accuracy for quantification. The combined information of XICs and isotope clusters can be referred to as a peptide LC-MS feature or more commonly simply as a feature.

Representation of LC-MS features in MasPy

Peptide LC-MS features, but also XICs, are represented in MasPy with the feature item class *`maspy.core.Fi`*. Its structure is kept very simple, similar to *`Si`* and *`Sii`*, with only a few mandatory attributes. Each instance is uniquely identified by the combination of the `Fi.id` and `Fi.specfile` attributes. However, the `id` attribute is not associated with any particular MS scan, which was the case for `Si` and `Sii`. Further attributes that should always be supplied when importing features into MasPy are `mz`, `rt`, `rtLow`, `rtHigh` and `intensity`. Altough not absolutely mandatory the `charge` attribute should also be supplied whenever possible, since the charge information is used in some algorithms.

The *`FiContainer`* is used to store feature items of one or multiple specfiles. The container allows saving and loading of imported results and provides methods for convenient access to the data.

Attribute naming conventions in MasPy and additional attributes that might be necessary for working with feature items:

#TODO: maybe change mz to obsMz to be consistent between data types

- `mz` the experimentally observed mass to charge ratio (Dalton / charge). Normally the m/z value of the monoisotopic peak.
- `rt` the retention time center of the feature.
- `rtLow` the lower retention time boundary of the feature.
- `rtHigh` the upper retention time boundary of the feature.
- `intensity` an estimator for the feature abundance. The preferred value is the integrated intensity area, but the feature apex intensity is also possible.
- `charge` the charge state of the feature.
- `peptide` the peptide sequence of the *`Sii`* that is used for annotating the feature.
- `sequence` the plain amino acid sequence of the *`Sii`* that is used for annotating the feature.
- `score` or any other score attribute name of the *`Sii`* that is used for annotating the feature. It describes the quality of a spectrum identifications.
- `obsMz` the experimentally observed mass to charge ratio of the feature (Dalton / charge). Normally the m/z value of the monoisotopic peak.
- `obsMh` the experimentally observed mass to charge ratio of the feature, calculated for the mono protonated ion (Dalton / charge). Normally the monoisotopic peak.

- `obsMass` the experimentally observed not protonated mass of a feature calculated by using the mz and charge values (Dalton / charge). Normally the monoisotopic mass.
- `excMz` the exact calculated mass to charge ratio of the peptide (Dalton / charge). Normally the monoisotopic ion.
- `excMh` the exact calculated mass to charge ratio of the peptide, calculated for the mono protonated state (Dalton / charge). Normally the monoisotopic ion.
- `excMass` the exact calculated mass of the not protonated peptide (Dalton / charge). Normally the monoisotopic mass.

MasPy internal feature item attributes:

- `isValid` can be used to flag if a Fi has passed a given quality threshold.
- `isMatched` can be used to flag if a Fi has been matched to any *`Si`* or *`Sii`* elements.
- `isAnnotated` can be used to flag if a Fi has been annotated with a *`Sii`* element and therefore with an identified peptide sequence.
- `siIds` a list of *`Si`* elements that have been matched to the feature item.
- `siiIds` a list of *`Sii`* elements that have been matched to the feature item.

Supported feature detection algorithms

Currently MasPy supports the import of two feature containing file types; the openMS feature file format `.featureXML` and the `.feature.tsv` format generated by the open source tool Dinosaur. However, adding import routines for additional file formats should be trivial an can be done on demand.

The FeatureFinderCentroided node from openMS is one of the best established open source LC-MS feature defining algorithms. It can be used independently of a data analysis pipeline and other processing steps. It was published in 2013 as part of a complete openMS pipeline: An Automated Pipeline for High- Throughput Label-Free Quantitative Proteomics. Since its publication it was applied in numerous publications and has been reused in at least two additional open source projects: DeMix and DeMix-Q.

Dinosaur: A Refined Open- Source Peptide MS Feature Detector published in 2016, is an algorithm based on the graph model concept for feature detection introduced by MaxQuant in 2008. Dinosaur seems to provide similar or better results then the FeatureFinderCentroided node of openMS with a substantial increase in runtime performance. It is available on Github.

Basic code examples

Importing peptide features

The function *`maspy.reader.importPeptideFeatures()`* is used to import LC- MS features from a file. It automatically recognises the file type by the file name extension and executes the respective import routine. Therefore the file extension has to be either `.featurexml` (openMS) or `.feature.tsv` (Dinosaur) and is not case sensitive. The imported feature items are stored in the `FiContainer` instance passed to the function.

```
import maspy.core
import maspy.reader

fiContainer = maspy.core.FiContainer()
maspy.reader.importPeptideFeatures(fiContainer, 'filelocation/f.featureXML',
                                   'specfile_name_1')
```

Matching spectrum identification items to feature items

The peptide underlying a LC-MS feature can be determined by using the information of identified MSn scans. In MasPy this can be achieved by using `maspy.featuremethods.matchToFeatures()`, which allows matching `Sii` to `Fi` elements by comparing their m/z, retention time and charge information. User defined tolerance values for matching should be passed to the function, for details see the docstring documentation. However, the default settings should be appropriate for typical high resolution MS1 data as obtained by Thermo Orbitrap instruments.

#TODO: describe the print output

```
>>> import maspy.featuremethods
>>> maspy.featuremethods.matchToFeatures(fiContainer, siiContainer,
>>>                                      specfiles='specfile_name_1')
------ specfile_name_1 ------
Annotated features:                      3802 / 20437 = 18.6 %
Spectra matched to features:             4240 / 4898 = 86.6 %
```

Note: #TODO: describe which attributes must be present in the Sii items and link to the tutorial that describes how to obtain these attributes. #charge, m/z, rentention time

Accessing data stored in a FiContainer

#TODO: describe .getItem(), .getArrays()

Fasta files in MasPy

The FASTA file format

The `fasta` format is the standard file format for storing protein sequence information in the field of protoeomics. A fasta file can store from only a few up to tens of thousands protein entries, for example when it contains the collection of all known protein coding genes of an organism.

Each protein entry begins with a headerline, which is followed by lines describing the amino acid sequence in the single letter code. A headerline always starts with the greater than symbol >, which dinstinguishes it from the sequence part. Headerlines can contain various amounts of information typically starting with an unique identifier of the entry followed for example by the protein name and a protein description. However, there are no common rules for the content as well as the syntax of the headerline. As a consequence each protein database might use its own fasta header format, which has to be known to allow a proper parsing of fasta files.

MasPy provides functions to import proteins from a fasta file and perform an in silico digestion thereof. For parsing of the fasta protein headers we rely on the Pyteomics library, which provides automatic recognition and parsing of the formats UniProtKB, UniRef, UniParc and UniMES (UniProt Metagenomic and Environmental Sequences), described at uniprot.org. In addition we have added a custom header parser for fasta files in the SGD format from yeastgenome.org, which can also serve as a template for additional parser functions, see `maspy.proteindb.fastaParseSgd()`.

Protein database in MasPy

Upon import parsed protein entries are represented as `maspy.proteindb.ProteinSequence` elements. They are stored in the `maspy.proteindb.ProteinDatabase` class and can be accessed with their unique ids in `ProteinDatabase.proteins`. Thereafter each protein sequence is digested in silico by using the specified cleavage rules, yielding smaller `maspy.proteindb.PeptideSequence` elements, which are stored in

ProteinDatabase.peptides and can be accessed with their amino acid sequence. Their proteins attribute contains a set() of protein ids and references all proteins that have generated the specific peptide sequence during digestion. In addition the peptide positions within the protein sequence are stored in the attribute proteinPositions. If the peptide was generated by only one protein, its isUnique attribute is set True. Each *ProteinSequence* contains a list of such unique and not unique peptides that were generated during the digestion, the attributes uniquePeptides and sharedPeptides respectively. In addition it also has an isUnique attribute, which is set True if the protein digestion has generated at least one unique peptide for a certain protein.

Basic code examples

Importing a fasta file

The function *maspy.proteindb.importProteinDatabase()* is used to import fasta files, perform an insilico digestion and return a ProteinDatabase instance. The parameters minLength and maxLength specify which peptides are stored in the ProteinDatabase after digestion. The default digestion rule is for a tryptic digestion, cutting c-terminally to Lysine and Arginine.

```
import maspy.proteindb
fastaPath = 'filelocation/human.fasta'
proteindb = maspy.proteindb.importProteinDatabase(fastaPath, minLength=7,
                                                  maxLength=50
                                                  )
```

By passing the headerParser argument it is possible to specify an alternative function which is called to interpret the fasta header line.

```
parser = maspy.proteindb.fastaParseSgd
proteindb = maspy.proteindb.importProteinDatabase(fastaPath,
                                                  headerParser=parser
                                                  )
```

If a fasta file contains decoy proteins or contamination proteins specifically marked by their unique ids starting with a certain tag it should be specified with the attributes decoyTag and contaminationTag respectively to allow a proper parsing of the fasta headers.

```
proteindb = maspy.proteindb.importProteinDatabase(fastaPath,
                                                  decoyTag='[decoy]',
                                                  contaminationTag='[cont]'
                                                  )
```

Specifying parameters for in silico digestion

The protein cleavage rule can be changed by passing a regular expression with the argument cleavageRule. The dictionary *maspy.constants.expasy_rules* contains cleavage rules of the most popular proteolytic enzymes. The concept for finding cleavage positions with regular expressions was adapted from the python library Pyteomics and the expasy_rules collection of cleavage rules was copied from pyteomics.parser.expasy_rules. The asp-n cleavage rule provides an example of an enzyme that cuts n-terminally of a specific amino acid.

```
>>> import maspy.constants
>>> aspN = maspy.constants.expasy_rules['asp-n']
>>> proteindb = maspy.proteindb.importProteinDatabase(fastaPath,
>>>                                                   cleavageRule=aspN
>>>                                                   )
>>> aspN
u'\\w(?=D)'
```

Besides defining the cleavage rule it is possible to specify the number of allowed missed cleavage positions, wheter protein n-terminal peptides should also be generated with the initial Methionine removed and if Leucine and Isoleucine should be treated as indistinguishable when assigning peptide sequences to proteins.

```
proteindb = maspy.proteindb.importProteinDatabase(fastaPath,
                                                   missedCleavage=2,
                                                   removeNtermM=True
                                                   ignoreIsoleucine=True
                                                   )
```

...

...

...

...

Depricated

The fasta format has become the standard format for storing protein sequences, where each amino acid is represented in the single letter code. “A sequence in FASTA format begins with a single-line description, followed by lines of sequence data. The description line is distinguished from the sequence data by a greater-than (“>”) symbol in the first column. The word following the “>” symbol is the identifier of the sequence, and the rest of the line is the description (both are optional). The sequence ends if another line starting with a “>” appears; this indicates the start of another sequence.” [https://en.wikipedia.org/wiki/FASTA_format]

CHAPTER 2

maspy package

Submodules

maspy.auxiliary module

A collection of helper functions used in different modules. Functions deal for example with saving files, encoding and decoding data in the json format, filtering of numpy arrays, data fitting, etc.

class `maspy.auxiliary.`**`DataFit`**(*dependentVarInput*, *independentVarInput*)

Bases: `object`

#TODO: docstring

Parameters

- **`splines`** – #TODO: docstring
- **`splineCycles`** – #TODO: docstring
- **`splineMinKnotPoins`** – #TODO: docstring
- **`splineOrder`** – #TODO: docstring
- **`splineInitialKnots`** – #TODO: docstring
- **`splineSubsetPercentage`** – #TODO: docstring
- **`splineTerminalExpansion`** – #TODO: docstring
- **`dependentVar`** – #TODO: docstring
- **`independentVar`** – #TODO: docstring

`corrArray`(*inputArray*)

#TODO: docstring

Parameters **`inputArray`** – #TODO: docstring

Returns #TODO docstring

generateSplines()
#TODO: docstring

processInput(*dataAveraging=False*, *windowSize=None*)
#TODO: docstring

Parameters

- **dataAveraging** – #TODO: docstring
- **windowSize** – #TODO: docstring

class maspy.auxiliary.**MaspyJsonEncoder**(*skipkeys=False*, *ensure_ascii=True*, *check_circular=True*, *allow_nan=True*, *sort_keys=False*, *indent=None*, *separators=None*, *default=None*)

Bases: `json.encoder.JSONEncoder`

Extension of the json.JSONEncoder to serialize MasPy classes.

Maspy classes need to define a _reprJSON() method, which returns a json serializable object.

default(*obj*)

Returns obj._reprJSON() if it is defined, else json.JSONEncoder.default(obj)

class maspy.auxiliary.**Memoize**(*function*)

Bases: `object`

A general memoization class, specify a function when creating a new instance of the class. The functions return value is returned and stored in `self.memo` when the instance is called with an argument for the first time. Later calls with the same argument return the cached value, instead of calling the function again.

Variables

- **function** – when `Memoize` is called this functions return value is returned.
- **memo** – a dictionary that records the `function` return values of already called variables.

class maspy.auxiliary.**PartiallySafeReplace**

Bases: `object`

Indirectly overwrite files by writing to temporary files and replacing them at once.

This is a context manager. When the context is entered, subsequently opened files will actually open temporary files. Each time the same file-path is opened, the same temporary file will be used.

When the context is closed, it will attempt to replace the original files with the content of the temporary files. Thus, several files can be prepared with less risk of data loss. Data loss is still possible if the replacement-operation fails (due to locking, not handled yet) or is interrupted.

open(*filepath*, *mode='w+b'*)
Opens a file - will actually return a temporary file but replace the original file when the context is closed.

maspy.auxiliary.**applyArrayFilters**(*array*, *posL*, *posR*, *matchMask*)
#TODO: docstring

Parameters

- **array** – #TODO: docstring
- **posL** – #TODO: docstring
- **posR** – #TODO: docstring
- **matchMask** – #TODO: docstring

Returns `numpy.array`, a subset of the input `array`.

`maspy.auxiliary.`**`averagingData`**(*array*, *windowSize=None*, *averagingType='median'*)
#TODO: docstring

Parameters

- **`array`** – #TODO: docstring
- **`windowSize`** – #TODO: docstring
- **`averagingType`** – "median" or "mean"

Returns #TODO: docstring

`maspy.auxiliary.`**`calcDeviationLimits`**(*value*, *tolerance*, *mode*)
Returns the upper and lower deviation limits for a value and a given tolerance, either as relative or a absolute difference.

Parameters

- **`value`** – can be a single value or a list of values if a list of values is given, the minimal value will be used to calculate the lower limit and the maximum value to calculate the upper limit
- **`tolerance`** – a number used to calculate the limits
- **`mode`** – either `absolute` or `relative`, specifies how the `tolerance` should be applied to the `value`.

`maspy.auxiliary.`**`factorial`** **= <maspy.auxiliary.Memoize object>**
Returns the factorial of a number, the results of already calculated numbers are stored in factorial.memo

`maspy.auxiliary.`**`findAllSubstrings`**(*string*, *substring*)
Returns a list of all substring starting positions in string or an empty list if substring is not present in string.

Parameters

- **`string`** – a template string
- **`substring`** – a string, which is looked for in the `string` parameter.

Returns a list of substring starting positions in the template string

`maspy.auxiliary.`**`joinpath`**(*path*, **paths*)
Join two or more pathname components, inserting "/" as needed and replacing all "" by "/".

Returns str

`maspy.auxiliary.`**`listFiletypes`**(*targetfilename*, *directory*)
Looks for all occurences of a specified filename in a directory and returns a list of all present file extensions of this filename.

In this cas everything after the first dot is considered to be the file extension: `"filename.txt" -> "txt"`, `"filename.txt.zip" -> "txt.zip"`

Parameters

- **`targetfilename`** – a filename without any extensions
- **`directory`** – only files present in this directory are compared to the targetfilename

Returns a list of file extensions (str)

`maspy.auxiliary.`**`loadBinaryItemContainer`**(*zippedfile*, *jsonHook*)
Imports binaryItems from a zipfile generated by *`writeBinaryItemContainer()`*.

Parameters

- **`zipfile`** – can be either a path to a file (a string) or a file-like object
- **`jsonHook`** – a custom decoding function for JSON formated strings of the binaryItems stored in the zipfile.

Returns a dictionary containing binaryItems `{binaryItem.id: binaryItem, ... }`

`maspy.auxiliary.`**`log10factorial`** **= <maspy.auxiliary.Memoize object>**

Returns the log10 factorial of a number, the results of already calculated numbers are stored in log10factorial.memo

`maspy.auxiliary.`**`matchingFilePaths`**(*targetfilename*, *directory*, *targetFileExtension=None*, *selector=None*)

Search for files in all subfolders of specified directory, return filepaths of all matching instances.

Parameters

- **`targetfilename`** – filename to search for, only the string before the last ”.” is used for filename matching. Ignored if a selector function is specified.
- **`directory`** – search directory, including all subdirectories
- **`targetFileExtension`** – string after the last ”.” in the filename, has to be identical if specified. ”.” in targetFileExtension are ignored, thus ”.txt” is treated equal to “txt”.
- **`selector`** – a function which is called with the value of targetfilename and has to return True (include value) or False (discard value). If no selector is specified, equality to targetfilename is used.

Returns list of matching file paths (str)

`maspy.auxiliary.`**`openSafeReplace`**(*filepath*, *mode='w+b'*)

Context manager to open a temporary file and replace the original file on closing.

`maspy.auxiliary.`**`returnArrayFilters`**(*arr1*, *arr2*, *limitsArr1*, *limitsArr2*)

#TODO: docstring

Parameters

- **`arr1`** – #TODO: docstring
- **`arr2`** – #TODO: docstring
- **`limitsArr1`** – #TODO: docstring
- **`limitsArr2`** – #TODO: docstring

Returns #TODO: docstring

`maspy.auxiliary.`**`returnSplineList`**(*dependentVar*, *independentVar*, *subsetPercentage=0.4*, *cycles=10*, *minKnotPoints=10*, *initialKnots=200*, *splineOrder=2*, *terminalExpansion=0.1*)

#TODO: docstring

Note: Expects sorted arrays.

Parameters

- **`dependentVar`** – #TODO: docstring
- **`independentVar`** – #TODO: docstring
- **`subsetPercentage`** – #TODO: docstring
- **`cycles`** – #TODO: docstring

- **`minKnotPoints`** – #TODO: docstring
- **`initialKnots`** – #TODO: docstring
- **`splineOrder`** – #TODO: docstring
- **`terminalExpansion`** – expand subsets on both sides

Returns #TODO: docstring

`maspy.auxiliary.`**`searchFileLocation`**(*targetFileName*, *targetFileExtension*, *rootDirectory*, *recursive=True*)

Search for a filename with a specified file extension in all subfolders of specified rootDirectory, returns first matching instance.

Parameters

- **`targetFileName`** (*`str`*) – #TODO: docstring
- **`rootDirectory`** (*`str`*) – #TODO: docstring
- **`targetFileExtension`** (*`str`*) – #TODO: docstring
- **`recursive`** – bool, specify whether subdirectories should be searched

Returns a filepath (str) or None

`maspy.auxiliary.`**`toList`**(*variable*, *types=(<class 'str'>, <class 'int'>, <class 'float'>)*)

Converts a variable of type string, int, float to a list, containing the variable as the only element.

Parameters **`variable`** (*`(str, int, float, others)`*) – any python object

Returns [variable] or variable

`maspy.auxiliary.`**`tolerantArrayMatching`**(*referenceArray*, *matchArray*, *matchTolerance*, *matchUnit*)

#TODO: docstring Note: arrays must be sorted

Parameters

- **`referenceArray`** – #TODO: docstring
- **`matchArray`** – #TODO: docstring
- **`matchTolerance`** – #TODO: docstring
- **`matchUnit`** – #TODO: docstring

Returns #TODO: docstring

#TODO: change matchUnit to "absolute", "relative" and remove the "*1e-6"

`maspy.auxiliary.`**`writeBinaryItemContainer`**(*filelike*, *binaryItemContainer*, *compress=True*)

Serializes the binaryItems contained in binaryItemContainer and writes them into a zipfile archive.

Examples of binaryItem classes are *`maspy.core.Ci`* and *`maspy.core.Sai`*. A binaryItem class has to define the function `_reprJSON()` which returns a JSON formated string representation of the class instance. In addition it has to contain an attribute `.arrays`, a dictionary which values are `numpy.array`, that are serialized to bytes and written to the `binarydata` file of the zip archive. See `_dumpArrayDictToFile()`

The JSON formated string representation of the binaryItems, together with the metadata, necessary to restore serialized numpy arrays, is written to the `metadata` file of the archive in this form: `[[serialized binaryItem, [metadata of a numpy array, ...]], ...]`

Use the method *`loadBinaryItemContainer()`* to restore a binaryItemContainer from a zipfile.

Parameters

- **`filelike`** – path to a file (str) or a file-like object
- **`binaryItemContainer`** – a dictionary containing binaryItems
- **`compress`** – bool, True to use zip file compression

`maspy.auxiliary.`**`writeJsonZipfile`** (*filelike*, *data*, *compress=True*, *mode='w'*, *name='data'*)
Serializes the objects contained in data to a JSON formated string and writes it to a zipfile.

Parameters

- **`filelike`** – path to a file (str) or a file-like object
- **`data`** – object that should be converted to a JSON formated string. Objects and types in data must be supported by the json.JSONEncoder or have the method `._reprJSON()` defined.
- **`compress`** – bool, True to use zip file compression
- **`mode`** – 'w' to truncate and write a new file, or 'a' to append to an existing file
- **`name`** – the file name that will be given to the JSON output in the archive

maspy.constants module

Contains frequently used variables and constants as for example the exact masses of atoms and amino acids or cleavage rules of the most common proteolytic enzymes.

`maspy.constants.`**`COMPOSITION`** **= <MagicMock name='mock.mass.Composition' id='140015506912648'>**
A Composition object stores a chemical composition of a substance. Basically, it is a dict object, with the names of chemical elements as keys and values equal to an integer number of atoms of the corresponding element in a substance.

The main improvement over dict is that Composition objects allow adding and subtraction. For details see `pyteomics.mass.Composition`.

`maspy.constants.`**`aaComp`** **= {'E': <MagicMock name='mock.mass.Composition()' id='140015505763464'>, 'I': <MagicMo**
A dictionary with elemental compositions of the twenty standard amino acid residues. This concept was inherited from `pyteomics.mass.std_aa_comp`.

`maspy.constants.`**`aaMass`** **= {}**
A dictionary with exact monoisotopic masses of the twenty standard amino acid residues. This concept was inherited from `pyteomics.mass.std_aa_comp`.

`maspy.constants.`**`aaModComp`** **= {'u:1': <MagicMock name='mock.mass.Composition()' id='140015505763464'>, 'u:1020'**
A dictionary with elemental compositions of the peptide modifications. Modifications present at `www.unimod.org` should be written as "u:X", where X is the unimod accession number. If a modification is not present in unimod a text abbriviation should be used. This concept was inherited from `pyteomics.mass.std_aa_comp`.

TODO: in the future this table should be imported from two external files. The first is directly obtained from www.unimod.org, the second contains user specified entries. It is also possible to specify a modification folder where multiple user specified files can be deposited for importing.

`maspy.constants.`**`aaModMass`** **= {}**
A dictionary with exact monoisotopic masses of peptide modifications.

`maspy.constants.`**`expasy_rules`** **= {'hydroxylamine': 'N(?=G)', 'trypsin': '([KR](?=[^P]))|((?<=W)K(?=P))|((?<=M)R(?**
The dictionary expasy_rules contains regular expressions for cleavage rules of the most popular proteolytic enzymes. The rules were copied from Pyteomics and initially taken from the PeptideCutter tool at Expasy.

maspy.core module

The core module contains python classes to represent spectra, peptide spectrum matches and peptide LC-MS features, and containers which manage storage, data access, saving and loading of these data types.

class `maspy.core.`**`Ci`**(*identifier*, *specfile*)

Bases: `object`

Chromatogram item (Ci), representation of a mzML `chromatogram`.

Variables

- ***`id`*** – The unique id of this chromatogram. Typically descriptive for the chromatogram, eg "TIC" (total ion current). Is used together with `self.specfile` as a key to access the spectrum in its container *`MsrunContainer.cic`*.
- ***`specfile`*** – An id representing a group of spectra, typically of the same mzML file / ms-run.
- ***`id`*** –
- ***`dataProcessingRef`*** – This attribute can optionally reference the 'id' of the appropriate dataProcessing, from mzML.
- ***`precursor`*** – The method of precursor ion selection and activation, from mzML.
- ***`product`*** – The method of product ion selection and activation in a precursor ion scan, from mzML.
- ***`params`*** – A list of parameter tuple, #TODO: as described elsewhere
- ***`arrays`*** – a dictionary containing the binary data of a chromatogram as `numpy.array`. Keys are derived from the specified mzML cvParam, see *`maspy.xml.findBinaryDataType()`*. Typically contains at least a time parameter `rt` (retention time) `Ci.arrays = {'rt': numpy.array(), ...}`
- ***`arrayInfo`*** – dictionary describing each data type present in `.arrays`.

```
{dataType: {'dataProcessingRef': str,
            'params': [paramTuple, paramTuple, ...]
            }
 }
```

code example:

```
{u'i': {u'dataProcessingRef': None,
        u'params': [('MS:1000521', '', None),
                    ('MS:1000574', '', None),
                    ('MS:1000515', '', 'MS:1000131')
                    ]
        },
 u'rt': {u'dataProcessingRef': None,
         u'params': [('MS:1000523', '', None),
                     ('MS:1000574', '', None),
                     ('MS:1000595', '', 'UO:0000031')
                     ]
         }
 }
```

`arrayInfo`

`arrays`

attrib

dataProcessingRef

id

static **jsonHook**(*encoded*)

Custom JSON decoder that allows construction of a new `Ci` instance from a decoded JSON object.

Parameters **encoded** – a JSON decoded object literal (a dict)

Returns "encoded" or one of the these objects: *Ci*, *MzmlProduct*, *MzmlPrecursor*

params

precursor

product

specfile

class maspy.core.**Fi**(*identifier*, *specfile*)

Bases: `object`

Feature item (Fi), representation of a peptide LC-MS feature.

Variables

- ***id*** – the unique identifier of a LC-MS feature, as generated by the software used for extracting features from MS1 spectra.
- ***specfile*** – An id representing an mzML file / ms-run filename.
- **rt** – a representative retention time value of the `Fi` (in seconds). For example the retention time of the feature apex.
- **mz** – a representative mass to charge value of the `Fi` (in Dalton / charge). For example the average m/z value of all data points.
- **charge** – the `Fi` charge state
- **intensity** – a meassure for the `Fi` abundance, used for relative quantification. Typically the area of the feature intensity over time.
- **isValid** – bool or None if not specified this attribute can be used to flag if a `Fi` has passed a given quality threshold. Can be used to filter valid elements in eg *FiContainer.getArrays()*.
- **isMatched** – bool or None if not specified True if any `Si` or `Sii` elements could be matched. Should be set to False on import.
- **isAnnotated** – bool or None if not specified True if any `Sii` elements could be matched. Should be set to False on import. Not sure yet how to handle annotation from other features.
- **siIds** – list of tuple(specfile, id) from matched *Si*
- **siiIds** – list of tuple(specfile, id) from matched *Sii*
- **peptide** – peptide sequence containing amino acid modifications. If multiple peptide sequences are possible due to multiple `Sii` matches the most likely must be chosen. A simple and accepted way to do this is by choosing the `Sii` identification with the best score.
- ***sequence*** – the plain amino acid sequence of `self.peptide`
- **bestScore** – the score of the acceppted `Sii` for annotation

static `jsonHook`(*encoded*)
Custom JSON decoder that allows construction of a new `Fi` instance from a decoded JSON object.

Parameters **`encoded`** – a JSON decoded object literal (a dict)

Returns "encoded" or *`Fi`*

class `maspy.core.`**`FiContainer`**
Bases: `object`

Conainer for *`Fi`* elements.

Variables

- **`container`** – contains the stored *`Fi`* elements.

```
{specfilename: {'Fi identifier': [Fi, ...], ...}
```

- **`info`** – a dictionary containing information about imported specfiles.

```
{specfilename: {'path': str},
 ...
 }
```

path: folder location used by the `FiContainer` to save and load data to the hard disk.

`addSpecfile`(*specfiles*, *path*)
Prepares the container for loading `fic` files by adding specfile entries to `self.info`. Use *`FiContainer.load()`* afterwards to actually import the files.

Parameters

- **`specfiles`** (*`str or [str, str, ..]`*) – the name of an ms-run file or a list of names
- **`path`** – filedirectory used for loading and saving `fic` files

`getArrays`(*attr=None*, *specfiles=None*, *sort=False*, *reverse=False*, *selector=None*, *defaultValue=None*)
Return a condensed array of data selected from *`Fi`* instances from `self.container` for fast and convenient data processing.

Parameters

- **`attr`** – list of *`Fi`* item attributes that should be added to the returned array. The attributes "id" and "specfile" are always included, in combination they serve as a unique id.
- **`defaultValue`** – if an item is missing an attribute, the "defaultValue" is added to the array instead.
- **`specfiles`** (*`str or [str, str, ..]`*) – filenames of ms-run files - if specified return only items from those files
- **`sort`** – if "sort" is specified the returned list of items is sorted according to the *`Fi`* attribute specified by "sort", if the attribute is not present the item is skipped.
- **`reverse`** – bool, set True to reverse sort order
- **`selector`** – a function which is called with each *Fi* item and has to return True (include item) or False (discard item). Default function is: `lambda si: True`. By default only items with `Fi.isValid == True` are returned.

Returns {'attribute1': numpy.array(), 'attribute2': numpy.array(), ... }

`getItem` (*specfile*, *identifier*)
Returns a *`Fi`* instance from `self.container`.

Parameters

- **`specfile`** – a ms-run file name
- **`identifier`** – item identifier `Fi.id`

Returns `self.container[specfile][identifier]`

`getItems` (*specfiles=None*, *sort=False*, *reverse=False*, *selector=None*)
Generator that yields filtered and/or sorted *`Fi`* instances from `self.container`.

Parameters

- **`specfiles`** (*`str or [str, str, ..]`*) – filenames of ms-run files - if specified return only items from those files
- **`sort`** – if "sort" is specified the returned list of items is sorted according to the *`Fi`* attribute specified by "sort", if the attribute is not present the item is skipped.
- **`reverse`** – bool, `True` reverses the sort order
- **`selector`** – a function which is called with each `Fi` item and has to return True (include item) or False (discard item). By default only items with `Fi.isValid == True` are returned.

Returns items from container that passed the selector function

`load` (*specfiles=None*)
Imports the specified `fic` files from the hard disk.

Parameters **`specfiles`** (*`None, str, [str, str]`*) – the name of an ms-run file or a list of names. If None all specfiles are selected.

`removeAnnotation` (*specfiles=None*)
Remove all annotation information from *`Fi`* elements.

Parameters **`specfiles`** (*`None, str, [str, str]`*) – the name of an ms-run file or a list of names. If None all specfiles are selected.

`removeSpecfile` (*specfiles*)
Completely removes the specified specfiles from the `FiContainer`.

Parameters **`specfiles`** – the name of an ms-run file or a list of names.

`save` (*specfiles=None*, *compress=True*, *path=None*)
Writes the specified specfiles to `fic` files on the hard disk.

Note: If `.save()` is called and no `fic` files are present in the specified path new files are generated, otherwise old files are replaced.

Parameters

- **`specfiles`** – the name of an ms-run file or a list of names. If None all specfiles are selected.
- **`compress`** – bool, True to use zip file compression
- **`path`** – filedirectory to which the `fic` files are written. By default the parameter is set to `None` and the filedirectory is read from `self.info[specfile]['path']`

setPath (*folderpath*, *specfiles=None*)
Changes the folderpath of the specified specfiles. The folderpath is used for saving and loading of `fic` files.

Parameters

- **specfiles** (`None, str, [str, str]`) – the name of an ms-run file or a list of names. If None all specfiles are selected.
- **folderpath** – a filedirectory

class `maspy.core.`**MsrunContainer**
Bases: `object`

Container for mass spectrometry data (eg MS1 and MS2 spectra), provides full support for mzML files, see mzML schema documentation.

Variables

- **rmc** – "run metadata container", contains mzML metadata elements from the mzML file as a `lxml.etree.Element` object. This comprises all `mzML` subelements, except for the *run*' element subelements `spectrumList` and `chromatogramList`.
- **cic** – "chromatogram item container", see `Ci`
- **smic** – "spectrum metadata item container", see `Smi`
- **saic** – "spectrum array item container", see `Sai`
- **sic** – "spectrum item container", see `Si`
- **info** – contains information about the imported specfiles.

```
{specfilename: {'path': str,
                'status': {u'ci': bool, u'rm': bool, u'sai': bool,
                           u'si': bool, u'smi': bool}
                },
 ...
 }
```

`path` contains information about the filelocation used for saving and loading msrun files in the maspy dataformat. `status` describes which datatypes are currently imported.

code example:

```
{u'JD_06232014_sample1_A': {u'path': u'C:/filedirectory',
                            u'status': {u'ci': True, u'rm': True,
                                        u'sai': True, u'si': True,
                                        u'smi': True
                                        }
                            }
 }
```

Note: The structure of the containers `rmc`, `cic`, `smic`, `saic` and `sic` is always: `{"specfilename": {"itemId": item, ...}, ...}`

addSpecfile (*specfiles*, *path*)
Prepares the container for loading `mrc` files by adding specfile entries to `self.info`. Use `MsrunContainer.load()` afterwards to actually import the files

Parameters

- **`specfiles`** (*`str or [str, str, ..]`*) – the name of an ms-run file or a list of names
- **`path`** – filedirectory used for loading and saving `mrc` files

`getArrays`(*attr=None*, *specfiles=None*, *sort=False*, *reverse=False*, *selector=None*, *defaultValue=None*)

Return a condensed array of data selected from *`Si`* instances from `self.sic` for fast and convenient data processing.

Parameters

- **`attr`** – list of *`Si`* item attributes that should be added to the returned array. The attributes "id" and "specfile" are always included, in combination they serve as a unique id.
- **`defaultValue`** – if an item is missing an attribute, the "defaultValue" is added to the array instead.
- **`specfiles`** (*`str or [str, str, ..]`*) – filenames of ms-run files, if specified return only items from those files
- **`sort`** – if "sort" is specified the returned list of items is sorted according to the *`Si`* attribute specified by "sort", if the attribute is not present the item is skipped.
- **`reverse`** – bool, set True to reverse sort order
- **`selector`** – a function which is called with each *`Si`* item and has to return True (include item) or False (discard item). Default function is: `lambda si:  True`

Returns {'attribute1': numpy.array(), 'attribute2': numpy.array(), ... }

`getItem`(*specfile*, *identifier*)

Returns a *`Si`* instance from `self.sic`.

Parameters

- **`specfile`** – a ms-run file name
- **`identifier`** – item identifier `Si.id`

Returns `self.sic[specfile][identifier]`

`getItems`(*specfiles=None*, *sort=False*, *reverse=False*, *selector=None*)

Generator that yields filtered and/or sorted *`Si`* instances from `self.sic`.

Parameters

- **`specfiles`** (*`str or [str, str, ..]`*) – filenames of ms-run files - if specified return only items from those files
- **`sort`** – if "sort" is specified the returned list of items is sorted according to the *`Si`* attribute specified by "sort", if the attribute is not present the item is skipped.
- **`reverse`** – bool, `True` reverses the sort order
- **`selector`** – a function which is called with each `Si` item and returns True (include item) or False (discard item). Default function is: `lambda si:  True`

Returns items from container that passed the selector function

`load`(*specfiles=None*, *rm=False*, *ci=False*, *smi=False*, *sai=False*, *si=False*)

Import the specified datatypes from `mrc` files on the hard disk.

Parameters

- **`specfiles`** (*`None, str, [str, str]`*) – the name of an ms-run file or a list of names. If None all specfiles are selected.

- **rm** – bool, True to import `mrc_rm` (run metadata)
- **ci** – bool, True to import `mrc_ci` (chromatogram items)
- **smi** – bool, True to import `mrc_smi` (spectrum metadata items)
- **sai** – bool, True to import `mrc_sai` (spectrum array items)
- **si** – bool, True to import `mrc_si` (spectrum items)

removeData(*specfiles=None*, *rm=False*, *ci=False*, *smi=False*, *sai=False*, *si=False*)
Removes the specified datatypes of the specfiles from the msrunContainer. To completely remove a specfile use *`MsrunContainer.removeSpecfile()`*, which also removes the complete entry from `self.info`.

Parameters

- **specfiles** (*None, str, [str, str]*) – the name of an ms-run file or a list of names. If None all specfiles are selected.
- **rm** – bool, True to select `self.rmc`
- **ci** – bool, True to select `self.cic`
- **smi** – bool, True to select `self.smic`
- **sai** – bool, True to select `self.saic`
- **si** – bool, True to select `self.sic`

removeSpecfile(*specfiles*)
Completely removes the specified specfiles from the `msrunContainer`.

Parameters specfiles (*str, [str, str]*) – the name of an ms-run file or a list of names. If None all specfiles are selected.

save(*specfiles=None*, *rm=False*, *ci=False*, *smi=False*, *sai=False*, *si=False*, *compress=True*, *path=None*)
Writes the specified datatypes to `mrc` files on the hard disk.

Note: If `.save()` is called and no `mrc` files are present in the specified path new files are generated, otherwise old files are replaced.

Parameters

- **specfiles** (*None, str, [str, str]*) – the name of an ms-run file or a list of names. If None all specfiles are selected.
- **rm** – bool, True to select `self.rmc` (run metadata)
- **ci** – bool, True to select `self.cic` (chromatogram items)
- **smi** – bool, True to select `self.smic` (spectrum metadata items)
- **sai** – bool, True to select `self.saic` (spectrum array items)
- **si** – bool, True to select `self.sic` (spectrum items)
- **compress** – bool, True to use zip file compression
- **path** – filedirectory to which the `mrc` files are written. By default the parameter is set to `None` and the filedirectory is read from `self.info[specfile]['path']`

setPath (*folderpath*, *specfiles=None*)
Changes the folderpath of the specified specfiles. The folderpath is used for saving and loading of `mrc` files.

Parameters

- **specfiles** (*None, str, [str, str]*) – the name of an ms-run file or a list of names. If None all specfiles are selected.
- **folderpath** – a filedirectory

class `maspy.core.`**MzmlPrecursor** (*spectrumRef=None*, *activation=None*, *isolationWindow=None*, *selectedIonList=None*, ***kwargs*)
Bases: `object`

MasPy representation of an mzML `Scan` element, see mzML schema documentation.

Variables

- ***spectrumRef*** – native id of the spectrum corresponding to the precursor spectrum
- ***activation*** – the mzML `activation` is represented as a tuple of param tuples. It is describing the type and energy level used for activation and should not be changed.
- ***isolationWindow*** – the mzML `isolationWindow` is represented as a tuple of parm tuples. It is describing the measurement and should not be changed.
- ***selectedIonList*** – a list of mzML `selectedIon` elements, which are represented as a tuple of param tuples.

Note: The attributes “sourceFileRef” and “externalSpectrumID” are not supported by MasPy on purpose, since they are only used to refere to scans which are external to the mzML file.

activation

isolationWindow

selectedIonList

spectrumRef

class `maspy.core.`**MzmlProduct** (*isolationWindow=None*, ***kwargs*)
Bases: `object`

MasPy representation of an mzML `Product` element, the mzML schema documentation does however not provide a lot of information how this element is intended to be used and which information can be present.

Variables ***isolationWindow*** – the mzML `isolationWindow` is represented as a tuple of parm tuples. It is describing the measurement and should not be changed.

isolationWindow

class `maspy.core.`**MzmlScan** (*scanWindowList=None*, *params=None*, ***kwargs*)
Bases: `object`

MasPy representation of an mzML `Scan` element, see mzML schema documentation.

Variables

- ***scanWindowList*** – a list of mzML `scanWindow` elements, which are represented as a tuple of parm tuples. The mzML `scanWindowList` is describing the measurement and should not be changed.

- ***params*** – a list of parameter tuple (cvParam tuple, userParam tuple or referencableParam-Group tuple) of an mzML `Scan` element.

Note: The attributes "sourceFileRef" and "externalSpectrumID" are not supported by MasPy on purpose, since they are only used to refere to scans which are external to the mzML file. The attribute "spectrumRef" could be included but seems kind of useless.

The attribute "instrumentConfigurationRef" should be included though: #TODO.

params

scanWindowList

class `maspy.core.`**`Sai`**(*identifier*, *specfile*)
Bases: `object`

Spectrum array item (Sai), representation of the binary data arrays of an mzML `spectrum`.

Variables

- ***id*** – The unique id of this spectrum, typically the scan number. Is used together with `self.specfile` as a key to access the spectrum in its container *`MsrunContainer.saic`*. Should be derived from the spectrums nativeID format (MS:1000767).
- ***specfile*** – An id representing a group of spectra, typically of the same mzML file / ms-run.
- ***arrays*** – a dictionary containing the binary data of the recorded ion spectrum as `numpy.array`. Keys are derived from the specified mzML cvParam, see *`maspy.xml.findBinaryDataType()`*. Must at least contain the keys `mz` (mass to charge ratio) and `i` (intensity). `Sai.arrays = {'mz': numpy.array(), 'i': numpy.array(), ...}`
- ***arrayInfo*** – dictionary describing each data type present in `.arrays`.

```
{dataType: {'dataProcessingRef': str,
            'params': [paramTuple, paramTuple, ...]
            }
 }
```

code example:

```
{u'i': {u'dataProcessingRef': None,
        u'params': [('MS:1000521', '', None),
                    ('MS:1000574', '', None),
                    ('MS:1000515', '', 'MS:1000131')]},
 u'mz': {u'dataProcessingRef': None,
         u'params': [('MS:1000523', '', None),
                     ('MS:1000574', '', None),
                     ('MS:1000514', '', 'MS:1000040')]}}
```

arrayInfo

arrays

id

static jsonHook(*encoded*)
Custom JSON decoder that allows construction of a new `Sai` instance from a decoded JSON object.

Parameters **encoded** – a JSON decoded object literal (a dict)

Returns "encoded" or `Sai`

`specfile`

class `maspy.core.`**`Si`**(*identifier*, *specfile*)

Bases: `object`

Spectrum item (Si) - this is the spectrum representation intended to be used in maspy. A simplified representation of spectrum metadata. Contains only specifically imported attributes, which are necessary for data analysis. Does not follow any PSI data structure or name space rules.

Additional attributes can be transferred from the corresponding `Smi` entry. This is done by default when importing an mzML file by using the function `maspy.reader.defaultFetchSiAttrFromSmi()`.

Variables

- ***`id`*** – The unique id of this spectrum, typically the scan number. Is used together with `self.specfile` as a key to access the spectrum in its container `MsrunContainer.sic`. Should be derived from the spectrums nativeID format (MS:1000767).
- ***`specfile`*** – An id representing a group of spectra, typically of the same mzML file / ms-run.
- **`isValid`** – bool, can be used for filtering.
- **`msLevel`** – stage of ms level in a multi stage mass spectrometry experiment.

static **`jsonHook`**(*encoded*)

Custom JSON decoder that allows construction of a new `Si` instance from a decoded JSON object.

Parameters **`encoded`** – a JSON decoded object literal (a dict)

Returns "encoded" or `Si`

class `maspy.core.`**`Sii`**(*identifier*, *specfile*)

Bases: `object`

Spectrum identification item (Sii) - representation of an MSn fragment spectrum annotation, also referred to as peptide spectrum match (PSM).

Variables

- ***`id`*** – The unique id of this spectrum, typically the scan number. Is used together with `self.specfile` as a key to access the spectrum in its container `SiiContainer` or the corresponding spectrum in a `MsrunContainer`.
- ***`specfile`*** – An id representing an mzML file / ms-run filename.
- **`rank`** – The rank of this `Sii` compared to others for the same MSn spectrum. The rank is based on a score defined in the `SiiContainer`. If multiple Sii have the same top score, they should all be assigned `self.rank = 1`.
- **`isValid`** – bool or None if not specified this attribute can be used to flag if a Sii has passed a given quality threshold or has been validated as correct. Is used to filter valid elements in eg `SiiContainer.getArrays()`.

static **`jsonHook`**(*encoded*)

Custom JSON decoder that allows construction of a new `Sii` instance from a decoded JSON object.

Parameters **`encoded`** – a JSON decoded object literal (a dict)

Returns "encoded" or `Sii`

class `maspy.core.`**`SiiContainer`**

Bases: `object`

Conainer for `Sii` elements.

Variables

- **`container`** – contains the stored `Sii` elements.

```
{specfilename: {'Sii identifier': [Sii, ...], ...}
```

- **`info`** – a dictionary containing information about imported specfiles.

```
{specfilename: {'path': str, 'qcAttr': str, 'qcLargerBetter': bool,
                'qcCutoff': float, 'rankAttr': str,
                'rankLargerBetter': bool
                },
 ...
 }
```

  path: folder location used by the `SiiContainer` to save and load data to the hard disk.

  qcAttr: name of the parameter to define a quality cutoff. Typically this is some sort of a global false positive estimator (eg FDR)

  qcLargerBetter: bool, True if a large value for the `.qcAttr` means a higher confidence.

  qcCutoff: float, the quality threshold for the specifed `.qcAttr`

  rankAttr: name of the parameter used for ranking `Sii` according to how well they match to a fragment ion spectrum, in the case when their are multiple `Sii` present for the same spectrum.

  rankLargerBetter: bool, True if a large value for the `.rankAttr` means a better match to the fragment ion spectrum

Note: In the future this container may be integrated in an evidence or an mzIdentML like container.

`addSiInfo`(*msrunContainer, specfiles=None, attributes=['obsMz', 'rt', 'charge']*)

Transfer attributes to `Sii` elements from the corresponding :class'Si' in `MsrunContainer.sic`. If an attribute is not present in the `Si` the attribute value in the `Sii``is set to ``None`.

Attribute examples: 'obsMz', 'rt', 'charge', 'tic', 'iit', 'ms1Id'

Parameters

- **`msrunContainer`** – an instance of `MsrunContainer` which has imported the corresponding specfiles
- **`specfiles`** – the name of an ms-run file or a list of names. If None all specfiles are selected.
- **`attributes`** – a list of `Si` attributes that should be transfered.

`addSpecfile`(*specfiles, path*)

Prepares the container for loading `siic` files by adding specfile entries to `self.info`. Use `SiiContainer.load()` afterwards to actually import the files.

Parameters

- **`specfiles`** (*`str or [str, str, ..]`*) – the name of an ms-run file or a list of names
- **`path`** – filedirectory used for loading and saving `siic` files

calcMz(*specfiles=None*, *guessCharge=True*, *obsMzKey='obsMz'*)
Calculate the exact mass for `Sii` elements from the `Sii.peptide` sequence.

Parameters

- **specfiles** – the name of an ms-run file or a list of names. If None all specfiles are selected.
- **guessCharge** – bool, True if the charge should be guessed if the attribute `charge` is missing from `Sii`. Uses the calculated peptide mass and the observed m/z value to calculate the charge.
- **obsMzKey** – attribute name of the observed m/z value in `Sii`.

getArrays(*attr=None*, *specfiles=None*, *sort=False*, *reverse=False*, *selector=None*, *defaultValue=None*)
Return a condensed array of data selected from `Sii` instances from `self.container` for fast and convenient data processing.

Parameters

- **attr** – list of `Sii` item attributes that should be added to the returned array. The attributes "id" and "specfile" are always included, in combination they serve as a unique id.
- **defaultValue** – if an item is missing an attribute, the "defaultValue" is added to the array instead.
- **specfiles** (*str or [str, str, ..]*) – filenames of ms-run files - if specified return only items from those files
- **sort** – if "sort" is specified the returned list of items is sorted according to the `Sii` attribute specified by "sort", if the attribute is not present the item is skipped.
- **reverse** – bool, set True to reverse sort order
- **selector** – a function which is called with each *Sii* item and has to return True (include item) or False (discard item). Default function is: `lambda si: True`. By default only items with `Sii.isValid == True` are returned.

Returns {'attribute1': numpy.array(), 'attribute2': numpy.array(), ... }

getItems(*specfiles=None*, *sort=False*, *reverse=False*, *selector=None*)
Generator that yields filtered and/or sorted `Sii` instances from `self.container`.

Parameters

- **specfiles** (*str or [str, str, ..]*) – filenames of ms-run files - if specified return only items from those files
- **sort** – if "sort" is specified the returned list of items is sorted according to the `Sii` attribute specified by "sort", if the attribute is not present the item is skipped.
- **reverse** – bool, `True` reverses the sort order
- **selector** – a function which is called with each `Sii` item and has to return True (include item) or False (discard item). By default only items with `Sii.isValid == True` are returned.

Returns items from container that passed the selector function

getValidItem(*specfile*, *identifier*)
Returns a `Sii` instance from `self.container` if it is valid, if all elements of `self.container[specfile][identifier] are ``Sii.isValid == False` then `None` is returned.

Parameters

- **`specfile`** – a ms-run file name
- **`identifier`** – item identifier `Sii.id`

Returns `Sii` or `None`

`load`(*specfiles=None*)

Imports `siic` files from the hard disk.

Parameters **`specfiles`** (*`None, str, [str, str]`*) – the name of an ms-run file or a list of names. If None all specfiles are selected.

`removeSpecfile`(*specfiles*)

Completely removes the specified specfiles from the `SiiContainer`.

Parameters **`specfiles`** – the name of an ms-run file or a list of names.

`save`(*specfiles=None*, *compress=True*, *path=None*)

Writes the specified specfiles to `siic` files on the hard disk.

Note: If `.save()` is called and no `siic` files are present in the specified path new files are generated, otherwise old files are replaced.

Parameters

- **`specfiles`** – the name of an ms-run file or a list of names. If None all specfiles are selected.
- **`compress`** – bool, True to use zip file compression
- **`path`** – filedirectory to which the `siic` files are written. By default the parameter is set to `None` and the filedirectory is read from `self.info[specfile]['path']`

`setPath`(*folderpath*, *specfiles=None*)

Changes the folderpath of the specified specfiles. The folderpath is used for saving and loading of `siic` files.

Parameters

- **`folderpath`** – a filedirectory
- **`specfiles`** (*`None, str, [str, str]`*) – the name of an ms-run file or a list of names. If None all specfiles are selected.

class `maspy.core.`**`Smi`**(*identifier*, *specfile*)

Bases: `object`

Spectrum metadata item (Smi), representation of all the metadata data of an mzML `spectrum`, excluding the actual binary data.

For details on the mzML `spectrum` element refer to the documentation,.

Variables

- ***`id`*** – The unique id of this spectrum, typically the scan number. Is used together with `self.specfile` as a key to access the spectrum in its container *`MsrunContainer.smic`*. Should be derived from the spectrums nativeID format (MS:1000767).
- ***`specfile`*** – An id representing a group of spectra, typically of the same mzML file / ms-run.

- ***attributes*** – dict, attributes of an mzML `spectrum` element
- ***params*** – a list of parameter tuple (cvParam tuple, userParam tuple or referencableParamGroup tuple) of an mzML `spectrum` element.
- ***scanListParams*** – a list of parameter tuple (cvParam tuple, userParam tuple or referencableParamGroup tuple) of an mzML `scanList` element.
- ***scanList*** – a list of *MzmlScan* elements, derived from elements of an an mzML `scanList` element.
- ***precursorList*** – a list of *MzmlPrecursor* elements, derived from elements of an an mzML `precursorList` element.
- ***productList*** – a list of *MzmlProduct* elements, derived from elements of an an mzML `productList` element.

> **Warning:** The `Smi` is used to generate `spectrum` xml elements by using the function *maspy.writer.xmlSpectrumFromSmi()*. In order to generate a valid mzML element all attributes of `Smi` have to be in the correct format. Therefore it is highly recommended to only use properly implemented and tested methods for making changes to any `Smi` attribute.

attributes

id

static jsonHook(*encoded*)

Custom JSON decoder that allows construction of a new `Smi` instance from a decoded JSON object.

Parameters encoded – a JSON decoded object literal (a dict)

Returns "encoded" or one of the these objects: *Smi*, *MzmlScan*, *MzmlProduct*, *MzmlPrecursor*

params

precursorList

productList

scanList

scanListParams

specfile

maspy.core.**addMsrunContainers**(*mainContainer*, *subContainer*)

Adds the complete content of all specfile entries from the subContainer to the mainContainer. However if a specfile of `subContainer.info` is already present in `mainContainer.info` its contents are not added to the mainContainer.

Parameters

- **mainContainer** – *MsrunContainer*
- **subContainer** – *MsrunContainer*

> **Warning:** does not generate new items, all items added to the `mainContainer` are still present in the `subContainer` and changes made to elements of one container also affects the elements of the other one (ie elements share same memory location).

maspy.errors module

#TODO: module description

exception `maspy.errors.`**`FileFormatError`**

Bases: `RuntimeError`

This exception is raised when a wrong file type is encountered.

maspy.featuremethods module

#TODO: module description

`maspy.featuremethods.`**`matchToFeatures`**(*fiContainer*, *specContainer*, *specfiles=None*, *fMassKey='mz'*, *sMassKey='obsMz'*, *isotopeErrorList=0*, *precursorTolerance=5*, *toleranceUnit='ppm'*, *rtExpansionUp=0.1*, *rtExpansionDown=0.05*, *matchCharge=True*, *scoreKey='pep'*, *largerBetter=False*)

Annotate *`Fi`* (Feature items) by matching *`Si`* (Spectrum items) or *`Sii`* (Spectrum identification items).

Parameters

- **`fiContainer`** – `maspy.core.FeatureContainer`, contains `Fi`.
- **`specContainer`** – *`maspy.core.MsrunContainer`* or *`maspy.core.SiiContainer`*, contains `Si` or `Sii`.
- **`specfiles`** (*`str, list or None`*) – filenames of ms-run files, if specified consider only items from those files
- **`fMassKey`** – mass attribute key in `Fi.__dict__`
- **`sMassKey`** – mass attribute key in `Si.__dict__` or `Sii.__dict__` (eg 'obsMz', 'excMz')
- **`isotopeErrorList`** (*`list or tuple of int`*) – allowed isotope errors relative to the spectrum mass, for example "0" or "1". If no feature has been matched with isotope error 0, the spectrum mass is increased by the mass difference of carbon isotopes 12 and 13 and matched again. The different isotope error values are tested in the specified order therefore "0" should normally be the first value of the list.
- **`precursorTolerance`** – the largest allowed mass deviation of `Si` or `Sii` relative to `Fi`
- **`toleranceUnit`** – defines how the `precursorTolerance` is applied to the mass value of `Fi`. `"ppm": mass * (1 +/- tolerance*1E-6)` or `"da": mass +/- value`
- **`rtExpansionUp`** – relative upper expansion of `Fi` retention time area. `limitHigh = Fi.rtHigh + (Fi.rtHigh - Fi.rtLow) * rtExpansionUp`
- **`rtExpansionDown`** – relative lower expansion of `Fi` retention time area. `limitLow = Fi.rtLow - (Fi.rtHigh - Fi.rtLow) * rtExpansionDown`
- **`matchCharge`** – bool, True if `Fi` and `Si` or `Sii` must have the same `charge` state to be matched.
- **`scoreKey`** – `Sii` attribute name used for scoring the identification reliability
- **`largerBetter`** – bool, True if higher score value means a better identification reliability

#TODO: this function is nested pretty badly and should maybe be rewritten #TODO: replace tolerance unit "ppm" by tolerance mode "relative" and change

repsective calculations

`maspy.featuremethods.`**`rtCalibration`**(*fiContainer*, *allowedRtDev=60*, *allowedMzDev=2.5*, *reference=None*, *specfiles=None*, *showPlots=False*, *plotDir=None*, *minIntensity=100000.0*)

Performs a retention time calibration between `FeatureItem` of multiple specfiles.

Variables

- **`fiContainer`** – Perform alignment on `FeatureItem` in `FeatureContainer.specfiles`
- **`allowedRtDev`** – maxium retention time difference of two features in two runs to be matched
- **`allowedMzDev`** – maxium relative m/z difference (in ppm) of two features in two runs to be matched
- **`showPlots`** – boolean, True if a plot should be generated which shows to results of the calibration
- **`plotDir`** – if not None and showPlots is True, the plots are saved to this location.
- **`reference`** – Can be used to specifically specify a reference specfile
- **`specfiles`** – Limit alignment to those specfiles in the fiContainer
- **`minIntensity`** – consider only features with an intensity above this value

maspy.inference module

The module "inference" is still in development. The interface of high and low level functions is not yet stable!

This module allows a simple protein inference; reporting a minimal set of proteins that can explain a set of peptides; defining shared und unique peptides; defining proteins that share equal evidence, protein subgroups and subsumable proteins. In addition it should provide an interface that allows convenient selection of observed peptides that can be used for quantification.

About protein groups In shotgun proteomics proteins are not directly analyzed, but via peptides generated by proteolytic digestion. Some of these peptides can't be unambiguously mapped to one single protein because of sequence homology between proteins.

As a consequence, it is often not possible to determine the exact set of proteins that have generated the observed peptides. The general aim of protein grouping is to group proteins with shared evidence together to reduce redundancy and simplify data analysis and interpretation. The result of protein grouping not only contains protein groups, but also information about the relation of proteins and the ambiguity of peptides. This also simplifies inspection of the observed evidence that indicate the presence of a protein.

There is however no consensus in the proteomics community how a protein group is defined. One approach is to group proteins together which observed peptide sets have a subset or sameset relation. Another possibility is to generate protein groups in a way, that they reflect the minimal number of proteins that can completely account for all observed peptides.

A mapping based protein grouping algorithm Protein groups, as generated by the function mappingBasedGrouping(), can be defined as a set of proteins, with sameset and subset relations. Each of these protein sets is essential to explain the presence of at least one peptide. In addition, proteins that share information with multiple groups, but are not a subset of any protein present in one of the groups, are assosicated with these groups as subsumable proteins.

Principle of the grouping algorithm:

- All samset proteins are merged and subset proteins are removed
- Each unique protein generates a new protein group. This group contains all sameset and subset proteins of the group leading proteins. All peptides mapped to these proteins are considered to be group peptides. (A unique protein is mapped to at least least one peptide that is mapped to only one protein.)
- Remaining peptides that are not explained by any group yet are seeds for the generation of new groups. All these peptides are still mapped to at least two proteins. The function _findRedundantProteins() generates a set of proteins that is considered as redundant. After removing these redundant proteins, all remaining peptides become unique. Proteins mapped to these peptides generate groups as described before.
- Proteins that share peptides with multiple groups, but are not a subset of the group leading proteins are associated as susumable proteins.

Protein grouping example data See unit test module ../tests/test_inference.py A grafical representation of the example data and the expected results, generated by the function mappingBasedGrouping(), can be found here: https://docs.google.com/drawings/d/1irbXDODwYsbrEw-Pa4rok5xguYrCbaqWl7F03qKGdC0/pub?w=3262&h=6566

Example import maspy.inference

proteinToPeptides = { '01_P1': {'01_p1', '01_p2'}, '02_P1': {'02_p1', '02_p2'}, '02_P2': {'02_p2'}, '03_P1': {'03_p1', '03_p2'}, '03_P2': {'03_p1', '03_p2'}, '04_P1': {'04_p1', '04_p2'}, '04_P2': {'04_p1', '04_p2'}, '04_P3': {'04_p2'}, '05_P1': {'05_p1', '05_p2', '05_p3'}, '05_P2': {'05_p1', '05_p4'}, '05_P3': {'05_p2', '05_p3', '05_p4'}, '06_P1': {'06_p1', '06_p2', '06_p3'}, '06_P2': {'06_p2', '06_p3'}, '06_P3': {'06_p2', '06_p4'}, '06_P4': {'06_p2', '06_p3', '06_p4'}, '06_P5': {'06_p2', '06_p4'}, '07_P1': {'07_p1', '07_p2'}, '07_P2': {'07_p1', '07_p3', '07_p4'}, '07_P3': {'07_p2', '07_p3'}, '07_P4': {'07_p3', '07_p5'}, '08_P1': {'08_p1', '08_p2'}, '08_P2': {'08_p3', '08_p4'}, '08_P3': {'08_p2', '08_p3'}, '09_P1': {'09_p1', '09_p3', '09_p4', '09_p5', '09_p7'}, '09_P2': {'09_p1', '09_p2'}, '09_P3': {'09_p2', '09_p3'}, '09_P4': {'09_p5', '09_p6'}, '09_P5': {'09_p6', '09_p7'}, '10_P1': {'10_p1', '10_p2'}, '10_P2': {'10_p3', '10_p4', '10_p5'}, '10_P3': {'10_p2', '10_p3'}, '10_P4': {'10_p2', '10_p3', '10_p4'}, }

inference = maspy.inference.mappingBasedGrouping(proteinToPeptides)

Todo

Hypothesis testing Add a private function that can be used to performs hypothesis tests. These tests comprise the evaluation of some general assumptions about the grouping results. - Group representatives should contain all group peptides (not included

are peptides from subsumable proteins).

- Peptides of a subsumable protein should not be fully covered by non- subsumable proteins of one single group.
- Group peptides of all the groups of a subsumable protein should contain all peptides of the respective subsumable protein.

function mappingBasedGrouping()

Refactoring

- Processing of individual clusters can be put into a function

- Processing of groupInitiatingProteins, redundantProteins and subsetProteins can be put into function
- The First part that defines various sets of proteins can be put into a function
- Finally remove asserts
- Maybe it is not necessary to work with merged proteins, after defining them...

Unittest

- Test if all clusters are properly stored: see inference.clusters and inference._proteinToClusterId

Note Whenever possible, we use the PSI-MS ontology terms to describe protein groups and protein relationships. However, in order to be less ambiguous some of the terms are more strictly defined in maspy or used in a slightly different context. The details of used terms are described below.

Protein cluster (proteinCluster) A protein cluster comprises all proteins that are somehow linked by shared peptide evidence. (related to PSI-MS MS:1002407)

Protein group (proteinGroup) A group of proteins that are essential to explain at least one peptide. A protein group consists of its leading, subset and subsumable proteins. However, subsumable proteins are more loosely associated with the group. The common peptide evidence of a group is the sum of all its leading and subset proteins. As a consequence, all leading and subset proteins must be a sameset or a subset of the protein groups common peptide evidence.

Representative protein (representative) Each protein group must have exactle one representative protein. This protein must be amongst the leading proteins of the group. (identical to PSI-MS MS:1002403)

Leading protein (leading) Each protein group must have at least one leading protein, to indicate proteins with the strongest evidence. (identical to PSI-MS MS:1002401) All proteins present in a protein group that are not leading proteins can be considered to be non-leading proteins (PSI-MS MS:1002402).

Same-set protein (sameset) A protein that shares exactly the same set of peptide evidence with one or multiple other proteins. (identical to PSI-MS MS:1001594)

Sub-set protein (subset) A protein with a sub-set of the peptide evidence of another protein. It can be a sub-set of multiple proteins, and those don't have to be in the same protein group. (identical to PSI-MS MS:1001596)

Subsumable protein (subsumable) A protein which peptide evidence is distributed across multiple proteins in at least two different protein groups. This term is mutually exclusiv with the term sub-set protein, and hierarchically below sub-set. That means that a protein can only be a subsumable if it is not already a subset of another protein. Also the protein is not allowed to be a "leading" protein in any protein group. (is related to the PSI-MS term MS:1002570 "multiply subsumable protein", but defined more strictly)

Relevant terms from the PSI-MS ontology

Group representative (MS:1002403) An arbitrary and optional flag applied to exactly one protein per group to indicate it can serve as the representative of the group, amongst leading proteins, in effect serving as a tiebreaker for approaches that require exactly one group representative.

Leading protein (MS:1002401) At least one protein within each group should be annotated as a leading protein to indicate it has the strongest evidence, or approximately equal evidence as other group members.

Non-leading protein (MS:1002402) Zero to many proteins within each group should be annotated as non-leading to indicate that other proteins have stronger evidence.

Cluster identifier (MS:1002407) An identifier applied to protein groups to indicate that they are linked by shared peptides.

Same-set protein (MS:1001594) A protein which is indistinguishable or equivalent to another protein, having matches to an identical set of peptide sequences.

Sequence sub-set protein (MS:1001596) A protein with a sub-set of the peptide sequence matches for another protein, and no distinguishing peptide matches.

Sequence subsumable protein (MS:1001598) A sequence same-set or sequence sub-set protein where the matches are distributed across two or more proteins.

Sequence multiply subsumable protein (MS:1002570) A protein for which the matched peptide sequences are the same, or a subset of, the matched peptide sequences for two or more other proteins combined. These other proteins need not all be in the same group.

Peptide unique to one protein (MS:1001363) A peptide matching only one.

Peptide shared in multiple proteins (MS:1001175) A peptide matching multiple proteins.

Additional terms used in maspy

Unique protein (uniqueProtein) A protein mapped to at least one unique peptide.

Super-set protein: A protein with a set of peptide matches, that is a strict superset of other proteins (strict means that the peptide matches are not equal).

class `maspy.inference.`**`Protein`**(*proteinId*, *peptides*)
Bases: `object`

Protein represents one entry from a protein database. It describes the protein relations and how a set of observed peptides are mapped to it.

Variables

- ***`id`*** – unique identifier of the protein, should allow mapping to a protein database
- **`peptides`** – a list of all peptides that are mapped onto a protein
- **`uniquePeptides`** – unique to one single protein (or one protein after merging sameset proteins)
- **`groupUniquePeptides`** – shared only with proteins of the same group
- **`groupSubsumablePeptides`** – shared only with subsumable proteins, but not with multiple other protein groups
- **`sharedPeptides`** – shared between proteins of multiple protein groups
- **`isSubset`** – set, protein is a subset of these proteins
- **`isSameset`** – set, protein set that shares identical evidence
- **`isLeading`** – bool, True if protein is a leading protein of its group.
- **`isSubsumable`** – bool, True if protein is associated with multiple protein groups as a subsumable protein.

class `maspy.inference.`**`ProteinGroup`**(*groupId*, *representative*)
Bases: `object`

A protein ambiguity group.

Variables

- `id` – unique identifier of the protein, should allow mapping to a protein database
- `representative` – a single protein that represents the group
- `proteins` – set, all leading proteins, subset proteins and the representative protein. Subsumable proteins are not included here!
- `leading` – set, leading proteins (MS:1002401)
- `subset` – set, sub-set proteins
- `subsumableProteins` – set, group sub-sumable proteins

addLeadingProteins (*proteinIds*)
Add one or multiple proteinIds as leading proteins.

Parameters `proteinIds` – a proteinId or a list of proteinIds, a proteinId must be a string.

addSubsetProteins (*proteinIds*)
Add one or multiple proteinIds as subset proteins.

Parameters `proteinIds` – a proteinId or a list of proteinIds, a proteinId must be a string.

addSubsumableProteins (*proteinIds*)
Add one or multiple proteinIds as subsumable proteins.

Parameters `proteinIds` – a proteinId or a list of proteinIds, a proteinId must be a string.

class `maspy.inference.`**ProteinInference** (*protToPeps*)
Bases: `object`

Contains the result of a protein inference analysis.

Variables

- `protToPeps` – dict, for each protein (=key) contains a set of associated peptides (=value). For Example {protein: {peptide, ...}, ...}
- `pepToProts` – dict, for each peptide (=key) contains a set of parent proteins (=value). For Example {peptide: {protein, ...}, ...}
- `proteins` – dict, protein ids pointing to Protein objects
- `groups` – dict, protein group ids pointing to ProteinGroup objects
- `clusters` – dict, cluster ids pointing to a list of protein group ids that are part of the same cluster. See “Cluster identifier (MS:1002407)”
- `_proteinToGroupIds` – dict, protein ids containing a set of protein group ids that they are associated with. NOTE: Protein objects do not contain a reference to their protein groups
- `_proteinToClusterId` – dict, protein ids containing a set of cluster ids that they are associated with. NOTE: Protein objects do not contain a reference to their clusters
- `_uniquePeptides` – set, all unique peptides of all protein groups. I.e. peptides that are only mapped to one protein (after merging same-set proteins).
- `_groupUniquePeptides` – set, all group unique peptides of all protein groups. I.e. peptides that are not unique but only explained by proteins of one single protein group and not by any subsumable proteins.
- `_groupSubsumablePeptides` – set, all group subsumable peptides of all protein groups. I.e. peptides that are only explained by proteins of one single protein group but also of subsumable proteins.

- **`_sharedPeptides`** – set, all shared peptides of all protein groups. I.e. peptides that are explained by proteins of multiple protein groups, not considering subsumable proteins.

`addLeadingToGroups` (*proteinIds*, *groupIds*)
Add one or multiple leading proteins to one or multiple protein groups.

Parameters

- **`proteinIds`** – a proteinId or a list of proteinIds, a proteinId must be a string.
- **`groupIds`** – a groupId or a list of groupIds, a groupId must be a string.

`addProteinGroup` (*groupRepresentative*)
Adds a new protein group and returns the groupId.

The groupId is defined using an internal counter, which is incremented every time a protein group is added. The groupRepresentative is added as a leading protein.

Parameters **`groupRepresentative`** – the protein group representing protein

Returns the protein groups groupId

`addSubsetToGroups` (*proteinIds*, *groupIds*)
Add one or multiple subset proteins to one or multiple protein groups.

Parameters

- **`proteinIds`** – a proteinId or a list of proteinIds, a proteinId must be a string.
- **`groupIds`** – a groupId or a list of groupIds, a groupId must be a string.

`addSubsumableToGroups` (*proteinIds*, *groupIds*)
Add one or multiple subsumable proteins to one or multiple protein groups.

Parameters

- **`proteinIds`** – a proteinId or a list of proteinIds, a proteinId must be a string.
- **`groupIds`** – a groupId or a list of groupIds, a groupId must be a string.

`getGroups` (*proteinId*)
Return a list of protein groups a protein is associated with.

`maspy.inference.`**`mappingBasedGrouping`** (*protToPeps*)
Performs protein grouping based only on protein to peptide mappings.

Parameters **`protToPeps`** – dict, for each protein (=key) contains a set of associated peptides (=value). For Example {protein: {peptide, ...}, ...}

#TODO: REFACTORING!!!

returns a ProteinInference object

maspy.isobar module

maspy.mit_stats module

This module contains functions to calculate a running mean, median and mode.

`maspy.mit_stats.`**`runningMean`** (*seq*, *N*, *M*)

Purpose: Find the mean for the points in a sliding window (fixed size)

as it is moved from left to right by one point at a time.

Inputs:

seq – list containing items for which a mean (in a sliding window) is

to be calculated (N items)

N – length of sequence M – number of items in sliding window

Otputs: means – list of means with size N - M + 1

`maspy.mit_stats.`**`runningMedian`**`(`*seq*, *M*`)`

Purpose: Find the median for the points in a sliding window (odd number in size)

as it is moved from left to right by one point at a time.

Inputs:

seq – list containing items for which a running median (in a sliding window)

is to be calculated

M – number of items in window (window size) – must be an integer > 1

Otputs:

medians – list of medians with size N - M + 1

Note:

1. The median of a finite list of numbers is the “center” value when this list is sorted in ascending order.
2. If M is an even number the two elements in the window that are close to the center are averaged to give the median (this is not by definition)

maspy.peptidemethods module

provides functions to work with peptide sequences, mass to charge ratios and modifications and calvulation of masses.

`maspy.peptidemethods.`**`calcMassFromMz`**`(`*mz*, *charge*`)`

Calculate the mass of a peptide from its mz and charge.

Parameters

- **`mz`** – float, mass to charge ratio (Dalton / charge)
- **`charge`** – int, charge state

Returns non protonated mass (charge = 0)

`maspy.peptidemethods.`**`calcMhFromMz`**`(`*mz*, *charge*`)`

Calculate the MH+ value from mz and charge.

Parameters

- **`mz`** – float, mass to charge ratio (Dalton / charge)
- **`charge`** – int, charge state

Returns mass to charge ratio of the mono protonated ion (charge = 1) `maspy.peptidemethods.`**`calcMzFromMass`**(*mass*, *charge*)

Calculate the mz value of a peptide from its mass and charge.

Parameters

- **`mass`** – float, exact non protonated mass
- **`charge`** – int, charge state

Returns mass to charge ratio of the specified charge state

`maspy.peptidemethods.`**`calcMzFromMh`**(*mh*, *charge*)

Calculate the mz value from MH+ and charge.

Parameters

- **`mh`** – float, mass to charge ratio (Dalton / charge) of the mono protonated ion
- **`charge`** – int, charge state

Returns mass to charge ratio of the specified charge state

`maspy.peptidemethods.`**`calcPeptideMass`**(*peptide*, ***kwargs*)

Calculate the mass of a peptide.

Parameters

- **`aaMass`** – A dictionary with the monoisotopic masses of amino acid residues, by default *`maspy.constants.aaMass`*
- **`aaModMass`** – A dictionary with the monoisotopic mass changes of modications, by default *`maspy.constants.aaModMass`*
- **`elementMass`** – A dictionary with the masses of chemical elements, by default `pyteomics.mass.nist_mass`
- **`peptide`** – peptide sequence, modifications have to be written in the format "[modificationId]" and "modificationId" has to be present in *`maspy.constants.aaModMass`*

#TODO: change to a more efficient way of calculating the modified mass, by first extracting all present modifications and then looking up their masses.

`maspy.peptidemethods.`**`digestInSilico`**(*proteinSequence*, *cleavageRule='[KR]'*, *missedCleavage=0*, *removeNtermM=True*, *minLength=5*, *maxLength=55*)

Returns a list of peptide sequences and cleavage information derived from an in silico digestion of a polypeptide.

Parameters

- **`proteinSequence`** – amino acid sequence of the poly peptide to be digested
- **`cleavageRule`** – cleavage rule expressed in a regular expression, see *`maspy.constants.expasy_rules`*
- **`missedCleavage`** – number of allowed missed cleavage sites
- **`removeNtermM`** – booo, True to consider also peptides with the N-terminal methionine of the protein removed
- **`minLength`** – int, only yield peptides with length >= minLength
- **`maxLength`** – int, only yield peptides with length <= maxLength

Returns

a list of resulting peptide enries. Protein positions start with `1` and end with `len(proteinSequence`.

```
[(peptide amino acid sequence,
  {'startPos': int, 'endPos': int, 'missedCleavage': int}
  ), ...
 ]
```

Note: This is a regex example for specifying N-terminal cleavage at lysine sites `\w(?=[K])`

`maspy.peptidemethods.`**`removeModifications`**(*peptide*)

Removes all modifications from a peptide string and return the plain amino acid sequence.

Parameters

- **`peptide`** – peptide sequence, modifications have to be written in the format "[modificationName]"
- **`peptide`** – str

Returns amino acid sequence of `peptide` without any modifications

`maspy.peptidemethods.`**`returnModPositions`**(*peptide*, *indexStart=1*, *removeModString='UNIMOD:'*)

Determines the amino acid positions of all present modifications.

Parameters

- **`peptide`** – peptide sequence, modifications have to be written in the format "[modificationName]"
- **`indexStart`** – returned amino acids positions of the peptide start with this number (first amino acid position = indexStart)
- **`removeModString`** – string to remove from the returned modification name

Returns {modificationName:[position1, position2, ...], ...}

#TODO: adapt removeModString to the new unimod ids in #maspy.constants.aaModComp ("UNIMOD:X" -> "u:X") -> also change unit tests.

maspy.proteindb module

The protein database module allows the import of protein sequences from fasta files, parsing of fasta entry headers and performing in silico digestion by specified cleavage rules to generate peptides.

class `maspy.proteindb.`**`PeptideSequence`**(*sequence*, *mc=None*)

Bases: `object`

Describes a peptide as derived by digestion of one or multiple proteins, can't contain any modified amino acids.

Parameters

- **`sequence`** – amino acid sequence of the peptide
- **`missedCleavage`** – number of missed cleavages, dependens on enzyme specificity
- **`proteins`** – protein ids that generate this peptide under certain digest condition
- **`proteinPositions`** – start position and end position of a peptide in a protein sequence. One based index, ie the first protein position is "1". `{proteinId:(startPosition, endPositions) ...}`

isUnique

static jsonHook(*encoded*)

Custom JSON decoder that allows construction of a new `PeptideSequence` instance from a decoded JSON object.

Parameters **encoded** – a JSON decoded object literal (a dict)

Returns "encoded" or *PeptideSequence*

length()

Returns the number of amino acids of the polypeptide sequence.

mass()

Returns the mass of the polypeptide sequence in Dalton.

missedCleavage

proteinPositions

proteins

sequence

class maspy.proteindb.**ProteinDatabase**

Bases: `object`

Describes proteins and peptides generated by an in silico digestion of proteins.

Variables

- **peptides** – {sequence:PeptideSequence(), ...} contains elements of *PeptideSequence* derived by an in silico digest of the proteins
- ***proteins*** – {proteinId:Protein(), proteinId:Protein()}, used to access *ProteinSequence* elements by their id
- **proteinNames** – {proteinName:Protein(), proteinName:Protein()}, alternative way to access *ProteinSequence* elements by their names. Must be populated manually
- **info** – a dictionary containing information about the protein database and parameters specified for the in silico digestion of the protein entries.

```
{'name': str, 'mc': str, 'cleavageRule': str, 'minLength': int
 'maxLength': int, 'ignoreIsoleucine': bool, 'removeNtermM': bool
 }
```

name: a descriptive name of the protein database, used as the file name when saving the protein database to the hard disk

mc: number of allowed missed cleavage sites **cleavageRule**: cleavage rule expressed in a regular expression **minLength**: minimal peptide length **maxLength**: maximal peptide length **ignoreIsoleucine**: if True Isoleucine and Leucinge in peptide

sequences are treated as indistinguishable.

removeNtermM: if True also peptides with the N-terminal Methionine of the protein removed are considered.

calculateCoverage()

Calcualte the sequence coverage masks for all protein entries.

For a detailed description see `_calculateCoverageMasks()`

classmethod **`load`**(*path*, *name*)
Imports the specified `proteindb` file from the hard disk.

Parameters

- **`path`** – filedirectory of the `proteindb` file
- **`name`** – filename without the file extension ".proteindb"

Note: this generates rather large files, which actually take longer to import than to newly generate. Maybe saving / loading should be limited to the protein database whitout in silico digestion information.

`save`(*path*, *compress=True*)
Writes the `.proteins` and `.peptides` entries to the hard disk as a `proteindb` file.

Note: If `.save()` is called and no `proteindb` file is present in the specified path a new files is generated, otherwise the old file is replaced.

Parameters

- **`path`** – filedirectory to which the `proteindb` file is written. The output file name is specified by `self.info['name']`
- **`compress`** – bool, True to use zip file compression

class `maspy.proteindb.`**`ProteinSequence`**(*identifier*, *sequence*, *name=''*)
Bases: `object`

Describes a protein.

Variables

- ***`id`*** – identifier of the protein, for example a uniprot id.
- **`name`** – name of the protein
- ***`sequence`*** – amino acid sequence of the protein
- **`fastaHeader`** – str(), the proteins faster header line
- **`fastaInfo`** – dict(), the interpreted fasta header as generated when using a faster header parsing function, see *`fastaParseSgd()`*.
- ***`isUnique`*** – bool, True if at least one unique peptide can be assigned to the protein
- **`uniquePeptides`** – a set of peptides which can be unambiguously assigned to this protein
- **`sharedPeptides`** – a set of peptides which are shared between different proteins
- **`coverageUnique`** – the number of amino acids in the protein sequence that are coverd by unique peptides
- **`coverageShared`** – the number of amino acids in the protein sequence that are coverd by unique or shared peptides

static **`jsonHook`**(*encoded*)
Custom JSON decoder that allows construction of a new `ProteinSequence` instance from a decoded JSON object.

Parameters **`encoded`** – a JSON decoded object literal (a dict)

Returns “encoded” or `ProteinSequence`

`length`()
Returns the mass of the polypeptide sequence in dalton.

`mass`()
Returns the number of amino acids of the polypeptide sequence.

`maspy.proteindb.`**`fastaParseSgd`**(*header*)
Custom parser for fasta headers in the SGD format, see www.yeastgenome.org.

Parameters **`header`** – str, protein entry header from a fasta file

Returns dict, parsed header

`maspy.proteindb.`**`importProteinDatabase`**(*filePath, proteindb=None, decoyTag='[decoy]', contaminationTag='[cont]', headerParser=None, forceId=False, cleavageRule='[KR]', minLength=5, maxLength=40, missedCleavage=2, ignoreIsoleucine=False, removeNtermM=True*)
Generates a `ProteinDatabase` by in silico digestion of proteins from a fasta file.

Parameters

- **`filePath`** – File path
- **`proteindb`** – optional an existing `ProteinDatabase` can be specified, otherwise a new instance is generated and returned
- **`decoyTag`** – If a fasta file contains decoy protein entries, they should be specified with a sequence tag
- **`contaminationTag`** – If a fasta file contains contamination protein entries, they should be specified with a sequence tag
- **`headerParser`** – optional a headerParser can be specified #TODO: describe how a parser looks like
- **`forceId`** – bool, if True and no id can be extracted from the fasta header the whole header sequence is used as a protein id instead of raising an exception.
- **`cleavageRule`** – cleavage rule expressed in a regular expression, see `maspy.constants.expasy_rules`
- **`missedCleavage`** – number of allowed missed cleavage sites
- **`removeNtermM`** – bool, True to consider also peptides with the N-terminal Methionine of the protein removed
- **`minLength`** – int, only yield peptides with length >= minLength
- **`maxLength`** – int, only yield peptides with length <= maxLength
- **`ignoreIsoleucine`** – bool, if True treat Isoleucine and Leucine in peptide sequences as indistinguishable

See also `maspy.peptidemethods.digestInSilico()`

maspy.reader module

This module provides functions to import various data types as maspy objects, which are associated with analysis workflows of mass spectrometry data. This currently comprises the mzML format, results of the percolator software and to some extent mzIdentML files, and file formats representing peptide LC-MS feature ".featureXML" and ".features.tsv".

`maspy.reader.`**`addSiiToContainer`**(*siiContainer*, *specfile*, *siiList*)

Adds the `Sii` elements contained in the siiList to the appropriate list in `siiContainer.container[specfile]`.

Parameters

- **`siiContainer`** – instance of *`maspy.core.SiiContainer`*
- **`specfile`** – unambiguous identifier of a ms-run file. Is also used as a reference to other MasPy file containers.
- **`siiList`** – a list of `Sii` elements imported from any PSM search engine results

`maspy.reader.`**`applySiiQcValidation`**(*siiContainer*, *specfile*)

Iterates over all Sii entries of a specfile in siiContainer and validates if they surpass a user defined quality threshold. The parameters for validation are defined in `siiContainer.info[specfile]`:

- `qcAttr`, `qcCutoff` and `qcLargerBetter`

In addition to passing this validation a `Sii` has also to be at the first list position in the `siiContainer.container`. If both criteria are met the attribute `Sii.isValid` is set to `True`.

Parameters

- **`siiContainer`** – instance of *`maspy.core.SiiContainer`*
- **`specfile`** – unambiguous identifier of a ms-run file. Is also used as a reference to other MasPy file containers.

`maspy.reader.`**`applySiiRanking`**(*siiContainer*, *specfile*)

Iterates over all Sii entries of a specfile in siiContainer and sorts Sii elements of the same spectrum according to the score attribute specified in `siiContainer.info[specfile]['rankAttr']`. Sorted Sii elements are then ranked according to their sorted position, if multiple Sii have the same score, all get the same rank and the next entries rank is its list position.

Parameters

- **`siiContainer`** – instance of *`maspy.core.SiiContainer`*
- **`specfile`** – unambiguous identifier of a ms-run file. Is also used as a reference to other MasPy file containers.

`maspy.reader.`**`ciFromXml`**(*xmlelement*, *specfile*)

`maspy.reader.`**`convertMzml`**(*mzmlPath*, *outputDirectory=None*)

Imports an mzml file and converts it to a MsrunContainer file

Parameters

- **`mzmlPath`** – path of the mzml file
- **`outputDirectory`** – directory where the MsrunContainer file should be written

if it is not specified, the output directory is set to the mzml files directory.

`maspy.reader.`**`defaultFetchSiAttrFromSmi`**(*smi*, *si*)

Default method to extract attributes from a spectrum metadata item (sai) and adding them to a spectrum item (si).

`maspy.reader.`**`fetchParentIon`**(*smi*)

`maspy.reader.`**`fetchScanInfo`**(*smi*) `maspy.reader.`**`fetchSpectrumInfo`**(*smi*)

`maspy.reader.`**`importMsgfMzidResults`**(*siiContainer*, *filelocation*, *specfile=None*, *qcAttr='eValue'*, *qcLargerBetter=False*, *qcCutoff=0.01*, *rankAttr='score'*, *rankLargerBetter=True*)

Import peptide spectrum matches (PSMs) from a MS-GF+ mzIdentML file, generate *`Sii`* elements and store them in the specified *`siiContainer`*. Imported `Sii` are ranked according to a specified attribute and validated if they surpass a specified quality threshold.

Parameters

- **`siiContainer`** – imported PSM results are added to this instance of *`siiContainer`*
- **`filelocation`** – file path of the percolator result file
- **`specfile`** – optional, unambiguous identifier of a ms-run file. Is also used as a reference to other MasPy file containers. If specified the attribute `.specfile` of all `Sii` is set to this value, else it is read from the mzIdentML file.
- **`qcAttr`** – name of the parameter to define a quality cut off. Typically this is some sort of a global false positive estimator (eg FDR)
- **`qcLargerBetter`** – bool, True if a large value for the `.qcAttr` means a higher confidence.
- **`qcCutOff`** – float, the quality threshold for the specifed `.qcAttr`
- **`rankAttr`** – name of the parameter used for ranking `Sii` according to how well they match to a fragment ion spectrum, in the case when their are multiple `Sii` present for the same spectrum.
- **`rankLargerBetter`** – bool, True if a large value for the `.rankAttr` means a better match to the fragment ion spectrum

For details on `Sii` ranking see *`applySiiRanking()`*

For details on `Sii` quality validation see *`applySiiQcValidation()`*

`maspy.reader.`**`importMzml`**(*filepath*, *msrunContainer=None*, *siAttrFromSmi=None*, *specfilename=None*)

Performs a complete import of a mzml file into a maspy MsrunContainer.

ParamsiAttrFromSmi allow here to specify a custom function that extracts params a from spectrumMetadataItem

Parameters **`specfilename`** – by default the filename will be used as the specfilename in the MsrunContainer and all mzML item instances, specify here an alternative specfilename to override the default one

`maspy.reader.`**`importPeptideFeatures`**(*fiContainer*, *filelocation*, *specfile*)

Import peptide features from a featureXml file, as generated for example by the OpenMS node featureFinderCentroided, or a features.tsv file by the Dinosaur command line tool.

Parameters

- **`fiContainer`** – imported features are added to this instance of `FeatureContainer`.
- **`filelocation`** – Actual file path
- **`specfile`** – Keyword (filename) to represent file in the `FeatureContainer`. Each filename can only occure once, therefore importing the same filename again is prevented.

`maspy.reader.`**`importPercolatorResults`**(*siiContainer*, *filelocation*, *specfile*, *psmEngine*, *qcAttr='qValue'*, *qcLargerBetter=False*, *qcCutoff=0.01*, *rankAttr='score'*, *rankLargerBetter=True*)

Import peptide spectrum matches (PSMs) from a percolator result file, generate *`Sii`* elements and store them in the specified *`siiContainer`*. Imported `Sii` are ranked according to a specified attribute and validated if they surpass a specified quality threshold.

Parameters

- **`siiContainer`** – imported PSM results are added to this instance of *`siiContainer`*
- **`filelocation`** – file path of the percolator result file
- **`specfile`** – unambiguous identifier of a ms-run file. Is also used as a reference to other MasPy file containers.
- **`psmEngine`** – PSM search engine used for peptide spectrum matching before percolator. For details see *`readPercolatorResults()`*. Possible values are 'comet', 'xtandem', 'msgf'.
- **`qcAttr`** – name of the parameter to define a quality cut off. Typically this is some sort of a global false positive estimator (eg FDR)
- **`qcLargerBetter`** – bool, True if a large value for the `.qcAttr` means a higher confidence.
- **`qcCutOff`** – float, the quality threshold for the specifed `.qcAttr`
- **`rankAttr`** – name of the parameter used for ranking `Sii` according to how well they match to a fragment ion spectrum, in the case when their are multiple `Sii` present for the same spectrum.
- **`rankLargerBetter`** – bool, True if a large value for the `.rankAttr` means a better match to the fragment ion spectrum

For details on `Sii` ranking see *`applySiiRanking()`*

For details on `Sii` quality validation see *`applySiiQcValidation()`*

`maspy.reader.`**`prepareSiiImport`**(*siiContainer*, *specfile*, *path*, *qcAttr*, *qcLargerBetter*, *qcCutoff*, *rankAttr*, *rankLargerBetter*)

Prepares the `siiContainer` for the import of peptide spectrum matching results. Adds entries to `siiContainer.container` and to `siiContainer.info`.

Parameters

- **`siiContainer`** – instance of *`maspy.core.SiiContainer`*
- **`specfile`** – unambiguous identifier of a ms-run file. Is also used as a reference to other MasPy file containers.
- **`path`** – folder location used by the `SiiContainer` to save and load data to the hard disk.
- **`qcAttr`** – name of the parameter to define a `Sii` quality cut off. Typically this is some sort of a global false positive estimator, for example a 'false discovery rate' (FDR).
- **`qcLargerBetter`** – bool, True if a large value for the `.qcAttr` means a higher confidence.
- **`qcCutOff`** – float, the quality threshold for the specifed `.qcAttr`
- **`rankAttr`** – name of the parameter used for ranking `Sii` according to how well they match to a fragment ion spectrum, in the case when their are multiple `Sii` present for the same spectrum.

- **rankLargerBetter** – bool, True if a large value for the .rankAttr means a better match to the fragment ion spectrum.

For details on Sii ranking see *applySiiRanking()*

For details on Sii quality validation see *applySiiQcValidation()*

maspy.reader.**readMsgfMzidResults**(*filelocation*, *specfile=None*)

Reads MS-GF+ PSM results from a mzIdentML file and returns a list of *Sii* elements.

Parameters

- **filelocation** – file path of the percolator result file
- **specfile** – optional, unambiguous identifier of a ms-run file. Is also used as a reference to other MasPy file containers. If specified all the .specfile attribute of all Sii are set to this value, else it is read from the mzIdentML file.

Returns [sii, sii, sii, ...]

maspy.reader.**readPercolatorResults**(*filelocation*, *specfile*, *psmEngine*)

Reads percolator PSM results from a txt file and returns a list of *Sii* elements.

Parameters

- **filelocation** – file path of the percolator result file
- **specfile** – unambiguous identifier of a ms-run file. Is also used as a reference to other MasPy file containers.
- **psmEngine** – PSM PSM search engine used for peptide spectrum matching before percolator. This is important to specify, since the scanNr information is written in a different format by some engines. It might be necessary to adjust the settings for different versions of percolator or the PSM search engines used.

  Possible values are 'comet', 'xtandem', 'msgf'.

Returns [sii, sii, sii, ...]

maspy.reader.**smiFromXmlSpectrum**(*xmlelement*, *specfile*)

maspy.writer module

Provides the possibility to write a new mzML file from an MsrunContainer instance, which is the maspy representation of a specfile.

maspy.writer.**writeMzml**(*specfile*, *msrunContainer*, *outputdir*, *spectrumIds=None*, *chromatogramIds=None*, *writeIndex=True*)

#TODO: docstring

Parameters

- **specfile** – #TODO docstring
- **msrunContainer** – #TODO docstring
- **outputdir** – #TODO docstring
- **spectrumIds** – #TODO docstring
- **chromatogramIds** – #TODO docstring

`maspy.writer.`**`xmlChromatogramFromCi`**(*index*, *ci*, *compression='zlib'*)
#TODO: docstring :param index: #TODO: docstring :param ci: #TODO: docstring :param compression: #TODO: docstring

Returns #TODO: docstring

`maspy.writer.`**`xmlGenBinaryDataArrayList`**(*binaryDataInfo*, *binaryDataDict*, *compression='zlib'*, *arrayTypes=None*)
#TODO: docstring

Params binaryDataInfo #TODO: docstring

Params binaryDataDict #TODO: docstring

Params compression #TODO: docstring

Params arrayTypes #TODO: docstring

Returns #TODO: docstring

`maspy.writer.`**`xmlGenPrecursorList`**(*precursorList*)
#TODO: docstring

Params precursorList #TODO: docstring

Returns #TODO: docstring

`maspy.writer.`**`xmlGenProductList`**(*productList*)
#TODO: docstring

Params productList #TODO: docstring

Returns #TODO: docstring

`maspy.writer.`**`xmlGenScanList`**(*scanList*, *scanListParams*)
#TODO: docstring

Params scanList #TODO: docstring

Params scanListParams #TODO: docstring

Returns #TODO: docstring

`maspy.writer.`**`xmlSpectrumFromSmi`**(*index*, *smi*, *sai=None*, *compression='zlib'*)
#TODO: docstring

Parameters

- **`index`** – The zero-based, consecutive index of the spectrum in the SpectrumList. (mzML specification)
- **`smi`** – a SpectrumMetadataItem instance
- **`sai`** – a SpectrumArrayItem instance, if none is specified no binaryDataArrayList is written
- **`compression`** – #TODO: docstring

Returns #TODO: docstring

maspy.sil module

The class LabelDescriptor allows the specification of a labeling strategy with stable isotopic labels. It can then be used to determine the labeling state of a given peptide and calculate the expected mass of alternative labeling states.

class `maspy.sil.`**`LabelDescriptor`**

Bases: `object`

Describes a MS1 stable isotope label setup for quantification.

Variables

- **`labels`** – Contains a dictionary with all possible label states, keys are increasing integers starting from 0, which correspond to the different label states.
- **`excludingModifictions`** – bool, True if any label has specified excludingModifications

`addLabel`(*aminoAcidLabels*, *excludingModifications=None*)

Adds a new labelstate.

Parameters

- **`aminoAcidsLabels`** – Describes which amino acids can bear which labels. Possible keys are the amino acids in one letter code and 'nTerm', 'cTerm'. Possible values are the modifications ids from *`maspy.constants.aaModMass`* as strings or a list of strings. An example for one expected label at the n-terminus and two expected

  labels at each Lysine: `{'nTerm': 'u:188', 'K': ['u:188', 'u:188']}`
- **`excludingModifications`** – optional, A Dectionary that describes which modifications can prevent the addition of labels. Keys and values have to be the modifications ids from *`maspy.constants.aaModMass`*. The key specifies the modification that prevents the label modification specified by the value. For example for each modification 'u:1' that is present at an amino acid or terminus of a peptide the number of expected labels at this position is reduced by one: `{'u:1':'u:188'}`

`maspy.sil.`**`expectedLabelPosition`**(*peptide*, *labelStateInfo*, *sequence=None*, *modPositions=None*)

Returns a modification description of a certain label state of a peptide.

Parameters

- **`peptide`** – Peptide sequence used to calculat the expected label state modifications
- **`labelStateInfo`** – An entry of `LabelDescriptor.labels` that describes a label state
- **`sequence`** – unmodified amino acid sequence of **:var:'peptide'**, if None it is generated by *`maspy.peptidemethods.removeModifications()`*
- **`modPositions`** – dictionary describing the modification state of "peptide", if None it is generated by *`maspy.peptidemethods.returnModPositions()`*

Returns

{sequence position: sorted list of expected label modifications on that position, ...

}

`maspy.sil.`**`modAminoacidsFromLabelInfo`**(*labelDescriptor*)

Returns a set of all amino acids and termini which can bear a label, as described in "labelDescriptor".

Parameters **`labelDescriptor`** – *`LabelDescriptor`* describes the label setup of an experiment

Returns #TODO: docstring

`maspy.sil.`**`modSymbolsFromLabelInfo`**(*labelDescriptor*)

Returns a set of all modiciation symbols which were used in the labelDescriptor

Parameters **`labelDescriptor`** – *`LabelDescriptor`* describes the label setup of an experiment

Returns #TODO: docstring

`maspy.sil.`**`returnLabelState`**(*peptide*, *labelDescriptor*, *labelSymbols=None*, *labelAminoacids=None*)
Calculates the label state of a given peptide for the label setup described in labelDescriptor

Parameters

- **`peptide`** – peptide which label state should be calcualted
- **`labelDescriptor`** – *`LabelDescriptor`*, describes the label setup of an experiment.
- **`labelSymbols`** – modifications that show a label, as returned by *`modSymbolsFromLabelInfo()`*.
- **`labelAminoacids`** – amino acids that can bear a label, as returned by *`modAminoacidsFromLabelInfo()`*.

Returns

integer that shows the label state: >=0: predicted label state of the peptide

-1: peptide sequence can't bear any labelState modifications -2: peptide modifications don't fit to any predicted labelState -3: peptide modifications fit to a predicted labelState, but not all

predicted labelStates are distinguishable

`maspy.sil.`**`returnLabelStateMassDifferences`**(*peptide*, *labelDescriptor*, *labelState=None*, *sequence=None*)
Calculates the mass difference for alternative possible label states of a given peptide. See also *`LabelDescriptor`*, *`returnLabelState()`*

Parameters

- **`peptide`** – Peptide to calculate alternative label states
- **`labelDescriptor`** – *`LabelDescriptor`* describes the label setup of an experiment
- **`labelState`** – label state of the peptide, if None it is calculated by *`returnLabelState()`*
- **`sequence`** – unmodified amino acid sequence of the "peptide", if None it is generated by *`maspy.peptidemethods.removeModifications()`*

Returns {alternativeLabelSate: massDifference, ...} or { } if the peptide label state is -1.

Note: The massDifference plus the peptide mass is the expected mass of an alternatively labeled peptide

maspy.xml module

#TODO: module description

class `maspy.xml.`**`MzmlReader`**(*mzmlPath*)
Bases: `object`

#TODO: docstring

Variables

- **mzmlPath** – #TODO: docstring
- **metadataNode** – #TODO: docstring
- **chromatogramList** – #TODO: docstring

loadMetadata()
#TODO: docstring

next()
#TODO: docstring

Returns #TODO: docstring

parseSpectra()
#TODO: docstring

Returns #TODO: docstring

maspy.xml.**binaryDataArrayTypes = {'MS:1000595': 'rt', 'MS:1000786': 'non-standard', 'MS:1000821': 'pressure', 'M**
#TODO: docstring

maspy.xml.**clearParsedElements**(*element*)
Deletes an element and all linked parent elements.

This function is used to save memory while iteratively parsing an xml file by removing already processed elements.

Parameters **element** – #TODO docstring

maspy.xml.**clearTag**(*tag*)
#TODO: docstring eg "{http://psi.hupo.org/ms/mzml}mzML" returns "mzML"

Parameters **tag** – #TODO docstring

Returns

maspy.xml.**cvParamFromDict**(*attributes*)
Python representation of a mzML cvParam = tuple(accession, value, unitAccession).

Parameters **attributes** – #TODO: docstring

Returns #TODO: docstring

maspy.xml.**decodeBinaryData**(*binaryData*, *arrayLength*, *bitEncoding*, *compression*)
Function to decode a mzML byte array into a numpy array. This is the inverse function of *encodeBinaryData()*. Concept inherited from `pymzml.spec.Spectrum._decode()` of the python library pymzML.

Parameters

- **binaryData** – #TODO: docstring
- **arrayLength** – #TODO: docstring
- **binEncoding** – #TODO: docstring
- **compression** – #TODO: docstring

Returns #TODO: docstring

maspy.xml.**encodeBinaryData**(*dataArray*, *bitEncoding*, *compression*)
Function to encode a `numpy.array` into a mzML byte array. This is the inverse function of *decodeBinaryData()*.

Parameters

- **dataArray** – #TODO: docstring
- **bitEncoding** – #TODO: docstring
- **compression** – #TODO: docstring

Returns #TODO: docstring

`maspy.xml.`**`extractBinaries`**(*binaryDataArrayList*, *arrayLength*)
#TODO: docstring

Parameters

- **binaryDataArrayList** – #TODO: docstring
- **arrayLength** – #TODO: docstring

Returns #TODO: docstring

`maspy.xml.`**`extractParams`**(*xmlelement*)
#TODO docstring

Parameters **xmlelement** – #TODO docstring

Returns #TODO docstring

`maspy.xml.`**`findBinaryDataType`**(*params*)
#TODO: docstring from: http://www.peptideatlas.org/tmp/mzML1.1.0.html#binaryDataArray a binaryDataArray "MUST supply a *child* term of MS:1000518 (binary data type) only once"

Parameters **params** – #TODO: docstring

Returns #TODO: docstring

`maspy.xml.`**`findParam`**(*params*, *targetValue*)
Returns a param entry (cvParam or userParam) in a list of params if its 'accession' (cvParam) or 'name' (userParam) matches the targetValue. return: cvParam, userParam or None if no matching param was found

Parameters

- **params** – #TODO: docstring
- **targetValue** – #TODO: docstring

Returns #TODO: docstring

`maspy.xml.`**`getParam`**(*xmlelement*)
Converts an mzML xml element to a param tuple.

Parameters **xmlelement** – #TODO docstring

Returns a param tuple or False if the xmlelement is not a parameter ('userParam', 'cvParam' or 'referenceableParamGroupRef')

`maspy.xml.`**`interpretBitEncoding`**(*bitEncoding*)
Returns a floattype string and a numpy array type.

Parameters **bitEncoding** – Must be either '64' or '32'

Returns (floattype, numpyType)

`maspy.xml.`**`recClearTag`**(*element*)
Applies maspy.xml.clearTag() to the tag attribute of the "element" and recursively to all child elements.

Parameters **element** – an **:instance:'xml.etree.Element'**

`maspy.xml.`**`recCopyElement`**(*oldelement*)
Generates a copy of an xml element and recursively of all child elements.

Parameters **`oldelement`** – an instance of lxml.etree._Element

Returns a copy of the "oldelement"

Warning: doesn't copy `.text` or `.tail` of xml elements

`maspy.xml.`**`recRemoveTreeFormating`**(*element*)

Removes whitespace characters, which are leftovers from previous xml formatting.

Parameters **`element`** – an instance of lxml.etree._Element

str.strip() is applied to the "text" and the "tail" attribute of the element and recursively to all child elements.

`maspy.xml.`**`refParamGroupFromDict`**(*attributes*)

Python representation of a mzML referencableParamGroup = ('ref', ref)

Parameters **`attributes`** – #TODO: docstring

Returns #TODO: docstring

Note: altough the mzML element referencableParamGroups is imported, its utilization is currently not implemented in MasPy.

`maspy.xml.`**`sublistReader`**(*xmlelement*)

#TODO: docstring

`maspy.xml.`**`userParamFromDict`**(*attributes*)

Python representation of a mzML userParam = tuple(name, value, unitAccession, type)

Parameters **`attributes`** – #TODO: docstring

Returns #TODO: docstring

`maspy.xml.`**`xmlAddParams`**(*parentelement*, *params*)

Generates new mzML parameter xml elements and adds them to the 'parentelement' as xml children elements.

Parameters

- **`parentelement`** – `xml.etree.Element`, an mzML element
- **`params`** – a list of mzML parameter tuples ('cvParam', 'userParam' or 'referencableParamGroup')

Module contents

CHAPTER 3

Indices and tables

- genindex
- modindex
- search

Python Module Index

Index