pymzML Documentation

Release 2.5.2

Koesters, M. and Fufezan, C.

Jan 01, 2023

CONTENTS

CHAPTER

ONE

CONTENTS

1.1 Introduction

1.1.1 General information

Module to parse mzML data in Python based on cElementTree

Contact information

Please refer to:

Dr. Christian Fufezan
Group Leader Experimental Bioinformatics
Cellzome GmbH
R&D Platform Technology & Science
GSK
Germany
eMail: christian@fufezan.net

https://fufezan.net

1.1.2 Summary

pymzML is an extension to Python that offers

- a) easy access to mass spectrometry (MS) data that allows the rapid development of tools
- b) a very fast parser for mzML data, the standard mass spectrometry data format
- c) a set of functions to compare and/or handle spectra
- d) random access in compressed files
- e) interactive data visualization

1.1.3 Implementation

pymzML requires Python3.7+. The module is freely available on pymzml.github.com or pypi, published under MIT license and only requires numpy and regex, however there are several optional dependencies for extended functionality like interactive plotting and deconvolution.

1.1.4 Download

Get the latest version via github

https://github.com/pymzml/pymzML

The complete Documentation can be found as pdf

http://pymzml.github.com/dist/pymzml.pdf

1.1.5 Citation

M Kösters, J Leufken, S Schulze, K Sugimoto, J Klein, R P Zahedi, M Hippler, S A Leidel, C Fufezan; pymzML v2.0: introducing a highly compressed and seekable gzip format, Bioinformatics, doi: https://doi.org/10.1093/bioinformatics/bty046

1.1.6 Installation

pymzML requires

Note: Consider to use a Python virtual environment for easy installation and use. Further, usage of python3.7+ is recommended.

Download pymzML using

- GitHub version: Start by cloning the GitHub repository:

```
user@localhost:~$ git clone https://github.com/pymzML/pymzml.git
user@localhost:~$ cd pymzml
user@localhost:~$ pip install -r requirements.txt
user@localhost:~$ python setup.py install
```

- pypi version:

```
user@localhost:~$ pip install pymzml # install standard version
user@localhost:~$ pip install "pymzml[plot]" # with plotting support
user@localhost:~$ pip install "pymzml[pynumpress]" # with pynumpress support
user@localhost:~$ pip install "pymzml[deconvolution]" # with deconvolution support␣
↪using ms_deisotope
user@localhost:~$ pip install "pymzml[full]" # full featured
```

If you have troubles installing the dependencies, install numpy first separately, since pynumpress requires numpy to be installed.

If you use Windows 7 please use the 'SDK7.1 command prompt' for installation of pymzML to assure correct compiling of the C extensions.

Testing

To test the package and correct installation:

```
tox
```

1.1.7 Contributing

Please read the contribution guidelines before contributing here

1.1.8 Code of Conduct

Since pymzML is an open source project maintained by the community, we established a code of conduct in order to facilitate an inclusive environment for all users, contributors and project memebers. Before contributing to pymzML, please read the code of conduct here

The latest Documentation was generated on: Jan 01, 2023

1.2 Quick Start

1.2.1 Parsing a mzML file and setting measured precision

simple_parser.**main**(*mzml_file*)

Basic example script to demonstrate the usage of pymzML. Requires a mzML file as first argument.

usage:

./simple_parser.py <path_to_mzml_file>

Note: This script uses the new syntax with the MS level being a property of the spectrum class (Spectrum.ms_level). The old syntax can be found in the script simple_parser_v2.py where the MS level can be queried as a key (Spectrum['ms level'])

```
#!/usr/bin/env python
import sys
import pymzml


def main(mzml_file):
    """
    Basic example script to demonstrate the usage of pymzML. Requires a mzML
    file as first argument.

    usage:

        ./simple_parser.py <path_to_mzml_file>

    Note:

        This script uses the new syntax with the MS level being a property of
        the spectrum class ( Spectrum.ms_level ). The old syntax can be found in
        the script simple_parser_v2.py where the MS level can be queried as a key
        (Spectrum['ms level'])


    """
    run = pymzml.run.Reader(mzml_file)
    for n, spec in enumerate(run):
        print(
            "Spectrum {0}, MS level {ms_level} @ RT {scan_time:1.2f}".format(
                spec.ID, ms_level=spec.ms_level, scan_time=spec.scan_time_in_minutes()
            )
        )
    print("Parsed {0} spectra from file {1}".format(n, mzml_file))
    print()


if __name__ == "__main__":
    if len(sys.argv) < 2:
        print(main.__doc__)
        exit()
    mzml_file = sys.argv[1]
    main(mzml_file)
```

1.2.2 Seeking in a mzML file

One of the features of pymzML is the ability to (create) and read indexed gzip which allows mzML file sizes to reach the levels of the original RAW format. The interface to random access into a mzML file is implemented by the magic get function in pymzMLs run class.

Alternatively, pymzML can also rapidly seek into any uncompressed mzML file, no matter if an index was included into the file or not.

```
#!/usr/bin/env python
import pymzml
```

(continues on next page)

(continued from previous page)

```
run = pymzml.run.Reader( 'tests/data/BSA1.mzML.gz' )
spectrum_with_id_2540 = run[ 2540 ]
```

1.2.3 Reading mzML indices with a custom regular expression

When reading mzML files with indices wich is not an integer or contains "scan=1" or similar, you can set a custom regex to parse the index when initializing the reader.

Say for example you have an index as in the example file Manuels_customs_ids.mzML:
<offset idRef="ManuelsCustomID=1 diesdas">4026</offset>

```
#!/usr/bin/env python
import pymzml
import re

index_re = re.compile(
    b'.*idRef="ManuelsCustomID=(?P<ID>.*) diesdas">(?P<offset>[0-9]*)</offset>'
)
run = pymzml.run.Reader(your_file_path, index_regex=index_re)
spec_1 = run[1]
```

The regular expression has to contain a group called ID and a group called offset. Also be aware that your regex need to be a byte string.

1.3 pymzML module content

1.3.1 Main reader interface

The class *Reader* has been designed to selectively extract data from a mzML file and to expose the data as a python object. Necessary information are read in and stored in a fast accessible format. The reader itself is an iterator, thus looping over all spectra follows the classical pythonian syntax. Additionally one can random access spectra by their nativeID if the file if not truncated by a conversion Program.

Note: The class `Writer` is still in development.

---

class pymzml.run.**Reader**(*path_or_file*, *MS_precisions=None*, *obo_version=None*, *build_index_from_scratch=False*, *skip_chromatogram=True*, *index_regex=None*, ***kwargs*)

Initialize Reader object for a given mzML file.

Parameters
path (*str*) – path to the mzml file to parse.

Keyword Arguments

- **MS_precisions** (*dict*) – measured precisions for the different MS levels. e.g.:

```
{
    1 : 5e-6,
    2 : 20e-6
}
```

- **obo_version** (*str, optional*) – obo version number as string. If not specified the version will be extracted from the mzML file

---

Note: Setting the precision for MS1 and MSn spectra has changed in version 1.2. However, the old syntax as kwargs is still compatible (e.g. 'MS1_Precision=5e-6').

__getitem__(*identifier*)

Access spectrum with native id 'identifier'.

Parameters
identifier (*str or int*) – last number in the id tag of the spectrum element

Returns
spectrum/chromatogram object with native id 'identifier'

Return type
spectrum (*Spectrum* or *Chromatogram*)

property file_class

Return file object in use.

get_chromatogram_count()

Number of chromatograms in file.

Returns
Number of chromatograms in file.

Return type
chromatogram count (int)

get_spectrum_count()

Number of spectra in file.

Returns
Number of spectra in file.

Return type
spectrum count (int)

next()

Function to return the next Spectrum element.

1.3.2 Spectrum and Chromatogram

The spectrum class offers a python object for mass spectrometry data. The spectrum object holds the basic information of the spectrum and offers methods to interrogate properties of the spectrum. Data, i.e. mass over charge (m/z) and intensity decoding is performed on demand and can be accessed via their properties, e.g. *peaks*.

The Spectrum class is used in the *Reader* class. There each spectrum is accessible as a spectrum object.

Theoretical spectra can also be created using the setter functions. For example, m/z values, intensities, and peaks can be set by the corresponding properties: *pymzml.spec.Spectrum.mz*, *pymzml.spec.Spectrum.i*, *pymzml.spec.Spectrum.peaks*.

Similar to the spectrum class, the chromatogram class allows interrogation with profile data (time, intensity) in an total ion chromatogram.

Spectrum

class pymzml.spec.**Spectrum**(*element=<Element '*>, *measured_precision=5e-06*)

Spectrum class which inherits from class *pymzml.spec.MS_Spectrum*

Parameters
element (*xml.etree.ElementTree.Element*) – spectrum as xml element

Keyword Arguments
measured_precision (*float*) – in ppm, i.e. 5e-6 equals to 5 ppm.

__getitem__(*accession*)

Access spectrum XML information by tag name

Parameters
accession (*str*) – name of the XML tag

Returns
value of the XML tag

Return type
value (float or str)

__add__(*other_spec*)

Adds two pymzml spectra

Parameters
other_spec (Spectrum) – spectrum to add to the current spectrum

Returns
reference to the edited spectrum

Return type
self (*Spectrum*)

Example:

```
>>> import pymzml
>>> s = pymzml.spec.Spectrum( measuredPrescision = 20e-6 )
>>> file_to_read = "../mzML_example_files/xy.mzML.gz"
>>> run = pymzml.run.Reader(
...     file_to_read ,
...     MS1_Precision = 5e-6 ,
...     MSn_Precision = 20e-6
```

(continues on next page)

(continued from previous page)

```
... )
>>> for spec in run:
...     s += spec
```

__sub__(*other_spec*)

Subtracts two pymzml spectra.

Parameters
other_spec (spec.Spectrum) – spectrum to subtract from the current spectrum

Returns
returns self after other_spec was subtracted

Return type
self (*spec.Spectrum*)

__mul__(*value*)

Multiplies each intensity with a float, i.e. scales the spectrum.

Parameters
value (*int, float*) – value to multiply the intensities with

Returns

returns self after intensities were scaled
by value

Return type
self (*spec.Spectrum*)

__truediv__(*value*)

Divides each intensity by a float, i.e. scales the spectrum.

Parameters
value (*int, float*) – value to divide the intensities by

Returns

returns self after intensities were scaled
by value.

Return type
self (*spec.Spectrum*)

property ID

Access the native id (last number in the id attribute) of the spectrum.

Returns
native ID of the spectrum

Return type
ID (str)

property TIC

Property to access the total ion current for this spectrum.

Returns
Total Ion Current of the spectrum.

Return type
TIC (float)

estimated_noise_level(*mode='median'*)

Calculates noise threshold for function remove_noise.

Different modes are available. Default is 'median'

Keyword Arguments
mode (*str*) – define mode for removing noise. Default = "median" (other modes: "mean", "mad")

Returns
estimate noise threshold

Return type
noise_level (float)

extreme_values(*key*)

Find extreme values, minimal and maximum m/z and intensity

Parameters
key (*str*) – m/z : "mz" or intensity : "i"

Returns
tuple of minimal and maximum m/z or intensity

Return type
extrema (tuple)

get(*acc*, *default=None*)

Mimic dicts get function.

Parameters

- **acc** (*str*) – accession or obo tag to return
- **default** (*None, optional*) – default value if acc is not found

has_overlapping_peak(*mz*)

Checks if a spectrum has more than one peak for a given m/z value and within the measured precision

Parameters
mz (*float*) – m/z value which should be checked

Returns
Returns `True` if a nearby peak is detected, otherwise `False`

Return type
Boolean (bool)

has_peak(*mz2find*)

Checks if a Spectrum has a certain peak. Requires a m/z value as input and returns a list of peaks if the m/z value is found in the spectrum, otherwise `[]` is returned. Every peak is a tuple of m/z and intensity.

Note: Multiple peaks may be found, depending on the defined precisions

Parameters
mz2find (*float*) – m/z value which should be found

Returns
list of m/z, i tuples

Return type
peaks (list)

Example:

```
>>> import pymzml
>>> example_file = 'tests/data/example.mzML'
>>> run = pymzml.run.Reader(
...     example_file,
...     MS_precisions =  {
...         1 : 5e-6,
...         2 : 20e-6
...     }
... )
>>> for spectrum in run:
...     if spectrum.ms_level == 2:
...             peak_to_find = spectrum.has_peak(1016.5404)
...             print(peak_to_find)
[(1016.5404, 19141.735187697403)]
```

highest_peaks(*n*)

Function to retrieve the n-highest centroided peaks of the spectrum.

Parameters
n (*int*) – number of highest peaks to return.

Returns
list mz, i tupls with n-highest

Return type
centroided peaks (list)

Example:

```
>>> run = pymzml.run.Reader(
...     "tests/data/example.mzML.gz",
...      MS_precisions =  {
...         1 : 5e-6,
...         2 : 20e-6
...     }
... )
>>> for spectrum in run:
...     if spectrum.ms_level == 2:
...         if spectrum.ID == 1770:
...             for mz,i in spectrum.highest_peaks(5):
...                print(mz, i)
```

property i

Returns the list of the intensity values. If the intensity values are encoded, the function `_decode()` is used to decode the encoded data.

The i property can also be set, e.g. for theoretical data. However, it is recommended to use the peaks property to set mz and intensity tuples at same time.

Returns
i (list): list of intensity values from the analyzed spectrum

property id_dict

Access to all entries stored the id attribute of a spectrum.

Returns
key value pairs for all entries in id attribute of a spectrum

Return type
id_dict (dict)

property index

Access the index of the spectrum.

Returns
index of the spectrum

Return type
index (int)

Note: This does not necessarily correspond to the native spectrum ID

---

property measured_precision

Sets the measured and internal precision

Returns
measured precision (e.g. 5e-6)

Return type
value (float)

property ms_level

Property to access the ms level.

Return type
ms_level (int)

property mz

Returns the list of m/z values. If the m/z values are encoded, the function `_decode()` is used to decode the encoded data. The mz property can also be set, e.g. for theoretical data. However, it is recommended to use the peaks property to set mz and intensity tuples at same time.

Returns
list of m/z values of spectrum.

Return type
mz (list)

peaks(*peak_type*)

Decode and return a list of mz/i tuples.

Parameters
peak_type (*str*) – currently supported types are: raw, centroided and reprofiled

Returns
list or numpy array of mz/i tuples or arrays

Return type
peaks (list or ndarray)

ppm2abs(*value*, *ppm_value*, *direction=1*, *factor=1*)

Returns the value plus (or minus, dependent on direction) the error (measured precision) for this value.

Parameters

- **value** (*float*) – m/z value
- **ppm_value** (*int*) – ppm value

Keyword Arguments

- **direction** (*int*) – plus or minus the considered m/z value. The argument *direction* should be 1 or -1
- **factor** (*int*) – multiplication factor for the imprecision. The argument *factor* should be bigger than 0

Returns
imprecision for the given value

Return type
imprecision (float)

property precursors

List the precursor information of this spectrum, if available. :returns: list of precursor ids for this spectrum. :rtype: precursor(list)

reduce(*peak_type='raw'*, *mz_range=(None, None)*)

Remove all m/z values outside the given range.

Parameters
mz_range (*tuple*) – tuple of min, max values

Returns
list of mz, i tuples in the given range.

Return type
peaks (list)

remove_noise(*mode='median'*, *noise_level=None*, *signal_to_noise_threshold=1.0*)

Function to remove noise from peaks, centroided peaks and reprofiled peaks.

Keyword Arguments

- **mode** (*str*) – define mode for removing noise. Default = "median"
- **modes** (*(other*) – "mean", "mad")

  noise_level (float): noise threshold signal_to_noise_threshold (float): S/N threshold for a peak to be accepted

Returns
Returns a list with tuples of m/z-intensity pairs above the noise threshold

Return type
reprofiled peaks (list)

property scan_time

Property to access the retention time and retention time unit. Please note, that we do not assume the retention time unit, if it is not correctly defined in the mzML. It is set to 'unicorns' in this case.

Returns
scan_time_unit (str):

Return type
scan_time (float)

`scan_time_in_minutes()`

Property to access the retention time in minutes. If the retention time unit is defined within the mzML, the retention time is converted into minutes and returned without the unit.

Return type
scan_time (float)

`property selected_precursors`

Property to access the selected precursors of a MS2 spectrum. Returns a list of dicts containing the precursors mz and, if available intensity and charge for each precursor.

Return type
selected_precursors (list)

`set_peaks(peaks, peak_type)`

Assign a custom peak array of type peak_type

Parameters

- **peaks** (*list or ndarray*) – list or array of mz/i values
- **peak_type** (*str*) – Either raw, centroided or reprofiled

`similarity_to(spec2, round_precision=0)`

Compares two spectra and returns cosine

Parameters
spec2 (Spectrum) – another pymzml spectrum that is compared to the current spectrum.

Keyword Arguments
round_precision (*int*) – precision mzs are rounded to, i.e. round(mz, round_precision)

Returns

value between 0 and 1, i.e. the cosine between the
two spectra.

Return type
cosine (float)

Note: Spectra data is transformed into an n-dimensional vector, where m/z values are binned in bins of 10 m/z and the intensities are added up. Then the cosine is calculated between those two vectors. The more similar the specs are, the closer the value is to 1.

`property t_mz_set`

Creates a set of integers out of transformed m/z values (including all values in the defined imprecision). This is used to accelerate has_peak function and similar.

Returns
set of transformed m/z values

Return type
t_mz_set (set)

transform_mz(*value*)

pymzml uses an internal precision for different tasks. This precision depends on the measured precision and is calculated when `spec.Spectrum.measured_precision` is invoked. transform_mz can be used to transform m/z values into the internal standard.

Parameters
value (*float*) – m/z value

Returns
to internal standard transformed mz value this value can be used to probe internal dictionaries, lists or sets, e.g. *pymzml.spec.Spectrum.t_mz_set()*

Return type
transformed value (float)

Example

```
>>> import pymzml
>>> run = pymzml.run.Reader(
...     "test.mzML.gz" ,
...     MS_precisions =  {
...         1 : 5e-6,
...         2 : 20e-6
...     }
... )
>>>
>>> for spectrum in run:
...     if spectrum.ms_level == 2:
...         peak_to_find = spectrum.has_deconvoluted_peak(
...             1044.5804
...         )
...         print(peak_to_find)
[(1044.5596, 3809.4356300564586)]
```

property transformed_mz_with_error

Returns transformed m/z value with error

Returns

Transformed m/z values in dictionary

{

m/z_with_error : [(m/z,intensity), . . .], . . .

}

Return type
tmz values (dict)

property transformed_peaks

m/z value is multiplied by the internal precision.

Returns
Returns a list of peaks (tuples of mz and intensity). Float m/z values are adjusted by the internal precision to integers.

Return type
Transformed peaks (list)

Chromatogram

class pymzml.spec.**Chromatogram**(*element*, *measured_precision=5e-06*, *param=None*)

Class for Chromatogram access and handling.

peaks()

Return the list of peaks of the spectrum as tuples (time, intensity).

Returns
list of time, intensity tuples

Return type
peaks (list)

Example:

```
>>> import pymzml
>>> run = pymzml.run.Reader(
...     spectra.mzMl.gz,
...     MS_precisions =  {
...         1 : 5e-6,
...         2 : 20e-6
...     }
... )
>>> for entry in run:
...     if isinstance(entry, pymzml.spec.Chromatogram):
...         for time, intensity in entry.peaks:
...             print(time, intensity)
```

Note: The peaks property can also be set, e.g. for theoretical data. It requires a list of time/intensity tuples.

property profile

Returns the list of peaks of the chromatogram as tuples (time, intensity).

Returns
list of time, i tuples

Return type
peaks (list)

Example:

```
>>> import pymzml
>>> run = pymzml.run.Reader(
...     spectra.mzMl.gz,
...     MS_precisions = {
...         1 : 5e-6,
...         2 : 20e-6
...     }
... )
>>> for entry in run:
```

(continues on next page)

(continued from previous page)

```
...     if isinstance(entry, pymzml.spec.Chromatogram):
...         for time, intensity in entry.peaks:
...             print(time, intensity)
```

Note: The peaks property can also be set, e.g. for theoretical data. It requires a list of time/intensity tuples.

property time

Returns the list of time values. If the time values are encoded, the function _decode() is used to decode the encoded data.

The time property can also be set, e.g. for theoretical data. However, it is recommended to use the profile property to set time and intensity tuples at same time.

Returns
list of time values from the analyzed chromatogram

Return type
time (list)

MS_Spectrum

class pymzml.spec.**MS_Spectrum**

General spectrum class for data handling.

get_element_by_name(*name*)

Get element from the original tree by it's unit name.

Parameters
name (*str*) – unit name of the mzml element.

Keyword Arguments
obo_version (*str, optional*) – obo version number.

get_element_by_path(*hooks*)

Find elements in spectrum by its path.

Parameters
hooks (*list*) – list of parent elements for the target element.

Returns
list of XML objects found in the path

Return type
elements (list)

Example

To access cvParam in scanWindow tag:

```
>>> spec.get_element_by_path(['scanList', 'scan', 'scanWindowList',
...     'scanWindow', 'cvParam'])
```

property measured_precision

Set the measured and internal precision.

Returns
measured Precision (e.g. 5e-6)

Return type
value (float)

to_string(*encoding='latin-1'*, *method='xml'*)

Return string representation of the xml element the spectrum was initialized with.

Keyword Arguments

- **encoding** (*str*) – text encoding of the returned string.

  Default is latin-1.

- **method** (*str*) – text format of the returned string.

  Default is xml, alternatives are html and text.

Returns
xml string representation of the spectrum.

Return type
element (str)

1.3.3 Utils

GSGW

class pymzml.utils.GSGW.**GSGW**(*file=None*, *max_idx=10000*, *max_idx_len=8*, *max_offset_len=8*, *output_path='./test.dat.igzip'*, *comp_str=-1*)

Generalized Gzip writer class with random access to indexed offsets.

Keyword Arguments

- **file** (*string*) – Filename for the resulting file
- **max_idx** (*int*) – max number of indices which can be saved in this file
- **max_idx_len** (*int*) – maximal length of the index in bytes, must be between 1 and 255
- **max_offset_len** (*int*) – maximal length of the offset in bytes
- **output_path** (*str*) – path to the output file

_allocate_index_bytes()

Allocate 'self.max_index_num' bytes of length 'self.max_idx_len' in the header for inserting the index later on.

_write_data(*data*)

Write data into file-stream.

Parameters
data (*str*) – uncompressed data

_write_gen_header(*Index=False*, *FLAGS=None*)

Write a valid gzip header with creation time, user defined flag fields and allocated index.

Keyword Arguments

- **Index** (*bool*) – whether to or not to write an index into this header.
- **FLAGS** (*list, optional*) – list of flags (FTEXT, FHCRC, FEXTRA, FNAME) to set for this header.

Returns
byte offset of the file pointer

Return type
offset (int)

_write_identifier(*identifier*)

Convert and write the identifier into output file.

Parameters
identifier (*str or int*) – identifier to write into index

_write_offset(*offset*)

Convert and write offset to output file.

Parameters
offset (*int*) – offset which will be formatted and written into file index

add_data(*data*, *identifier*)

Create a new gzip member with compressed 'data' indexed with 'index'.

Parameters

- **data** (*str*) – uncompressed data to write to file
- **index** (*str or int*) – unique index for the data

property encoding

Returns the encoding used for this file

property file_out

Output filehandler

write_index()

Only called after all the data is written, i.e. all calls to *add_data()* have been done.

Seek back to the beginning of the file and write the index into the allocated comment bytes (see _write_gen_header(Index=True)).

GSGR

class pymzml.utils.GSGR.**GSGR**(*file=None*)

Generalized Gzip reader class which enables random access in files written with the *GSGW* class.

Keyword Arguments
file (*str*) – path to file to read

_check_magic_bytes()

Check if file is a gzip file.

_read_basic_header()

Read and save compression method, bitflags, changetime, compression speed and os.

_read_index()

Read and save offset dict from indexed gzip file

read(*size=-1*)

Read the content of the in File in binary mode

Keyword Arguments
size (*int, optional*) – number of bytes to read, -1 for everything

Returns
parsed bytes from input file

Return type
data (bytes)

read_block(*index*)

Read and return the data block with the unique index *index*

Parameters
index (*int or str*) – identifier associated with a specific block

Returns
indexed text block as string

Return type
data (str)

seek(*offset*)

Seek to byte offset in input file.

Parameters
offset (*int*) – byte offset to seek to in FileIn

Returns
None

Example:

```
..      class SQLiteDatabase(object):
..              """
..              Example implementation of a database Connector,
..              which can be used to make run accept paths to
..              sqlite db files.
```

Example:

```
..      def _open(self, path):
..              if path.endswith('.gz'):
..                      if self._indexed_gzip(path):
..                              self.file_handler = indexedGzip.IndexedGzip(path, self.
↪encoding)
..                      else:
..                              self.file_handler = standardGzip.StandardGzip(path, self.
↪encoding)
..              # Insert a new condition to enable your new fileclass
..              elif path.endswith('.db'):
..                      self.file_handler = utils.SQLiteConnector.SQLiteDatabase(path,␣
↪self.encoding)
..              else:
..                      self.file_handler     = standardMzml.StandardMzml(path, self.
↪encoding)
..              return self.file_handler
```

1.3.4 Plotting functions

Plotting functions for pymzML. The Factory object can hold several plots with several data traces each. The data can be visualized as an interactive plotly plot or be exported as JSON.

class pymzml.plot.**Factory**(*filename=None*)

__init__(*filename=None*)

Interface to visualize m/z or profile data using plotly (https://plot.ly/).

Parameters

filename (*str*) – Name for the output file. Default = "spectrum_plot.html"

add(*data*, *color=(0, 0, 0)*, *style='sticks'*, *mz_range=None*, *int_range=None*, *opacity=0.8*, *dash='solid'*, *name=None*, *plot_num=-1*, *title=None*)

Add data to the graph.

Parameters

data (*list*) – The data added to the graph. Must be list of tuples with the following format. Note that i can be set to 'max_intensity' for setting the label position to the maximum intensity.

- (mz,i) for all styles, except label,
- (mz,i, string) for label.hoverinfo, .sticks and .triangle
- (mz1, mz2, i, string) for label.linear and .spline

Keyword Arguments

- **color** (*tuple*) – color encoded in RGB. Default = (0,0,0)
- **style** (*str*) – plotting style. Default = "sticks". Currently supported styles are:
  - 'lines': Datapoints connected by lines
  - 'points': Datapoints without connection
  - 'sticks': Vertical line at given m/z (corresponds to centroided peaks)
  - 'triangle' (MS_precision, micro, tiny, small, medium, big): The top of the triangle corresponds to the given m/z, the width corresponds to he given format, e.g. 'triangle.MS_precision'

- 'label.hoverinfo': Label string appears as plotly hover info
- 'label.linear' (top, medium or bottom)
- 'label.spline' (top, medium or bottom)
- 'label.sticks'
- 'label.triangle' (small, medium or big)

- **mz_range** (*list*) – Boundaries that should be added to the current plot
- **int_range** (*list*) – Boundaries that should be added to the current plot
- **opacity** (*float*) – opacity of the data points
- **dash** (*str*) – type of line ('solid', 'dash', 'longdash', 'dot', 'dashdot', 'longdashdot')
- **name** (*str*) – name of data in legend
- **plot_num** (*int*) – Add data to plot[plot_num]
- **title** (*str*) – an optional title that will be printed above the plot

Note: The mz_range and int_range in the add() function sets the limits of datapoints added to the plot. This is in contrast to defining a range in the layout, which only defines the area that is depicted (i.e. the zoom) but still adds the datapoints to the plot (can be seen by zooming out).

get_data()

Return data and layout in JSON format.

Returns
JSON compatible python dict

Return type
plots (dict)

info()

Prints summary about the plotting factory, i.e. how many plots and how many datasets per plot.

newPlot(*MS_precision='5e-6'*, *title=None*)

Deprecated since version 1.2.

new_plot(*MS_precision='5e-6'*, *title=None*)

Add new plot to the plotting Factory. Every plot will be put into the x * 2 grid of one single figure.

Keyword Arguments

- **title** (*str*) – an optional title that will be printed above the plot
- **MS_precision** (*float*) – measuring MS_precision used in handler. Default 5e-6.

Note: Old function newPlot() is still working. However, the new syntax should be used.

save(*filename=None*, *mz_range=None*, *int_range=None*, *layout=None*, *write_pdf=False*)

Saves all plots and their data points that have been added to the plotFactory.

Keyword Arguments

- **filename** (*str*) – Name for the output file. Default = "spectrum_plot.html"
- **mz_range** (*list*) – m/z range which should be considered [start, end]. Default = None

- **int_range** (*list*) – intensity range which should be considered [min, max]. Default = None
- **layout** (*dict*) – dictionary containing layout information in plotly style
- **write_pdf** (*bool*) – Set "True" in order to save plots as pdf file (on Unix systems, this requires Orca to be installed)

Note: mz_range and int_range defined here will be applied to all subplots in the current plot, i.e. ranges defined when adding a subplot will be overwritten. To avoid this, a list of lists can be given in the order corresponding to the subplots.

1.3.5 File Access

pymzML offers support for different kinds of mzML files. The following classes are wrappers for access of different types of mzML files, which allows the implementation of file type specific search and data retrieving algorithms. An explanation of how to implement your own file class can be found in the advanced usage section.

File Interface

class pymzml.file_interface.**FileInterface**(*path*, *encoding*, *build_index_from_scratch=False*, *index_regex=None*)

Interface to different mzML formats.

__getitem__(*identifier*)

Access the item with id 'identifier' in the file.

Parameters
identifier (*str*) – native id of the item to access

Returns
text associated with the given identifier

Return type
data (str)

__init__(*path*, *encoding*, *build_index_from_scratch=False*, *index_regex=None*)

Initialize a object interface to mzML files.

Parameters

- **path** (*str*) – path to the mzML file
- **encoding** (*str*) – encoding of the file

_indexed_gzip(*path*)

Check if the given file is an indexed gzip file or not.

Parameters
path (*str*) – path to the file

Returns
True if path is a gzip file with index, else *False*

Return type
bool

_open(*path_or_file*)

Open a file like object resp. a wrapper for a file like object.

Parameters
path (*str*) – path to the mzml file

Returns
instance of *StandardGzip*, *IndexedGzip* or *StandardMzml*, based on the file ending of 'path'

Return type
file_handler

read(*size=-1*)

Read binary data from file handler.

Keyword Arguments

- **size** (*int*) – Number of bytes to read from file, -1 to
- **file** (*read to end of*) –

Returns
byte string with defined size of the input data

Return type
data (str)

File Classes

mzML

class pymzml.file_classes.standardMzml.**StandardMzml**(*path*, *encoding*, *build_index_from_scratch=False*, *index_regex=None*)

__getitem__(*identifier*)

Access the item with id 'identifier'.

Either use linear, binary or interpolated search.

Parameters
identifier (*str*) – native id of the item to access

Returns
text associated with the given identifier

Return type
data (str)

__init__(*path*, *encoding*, *build_index_from_scratch=False*, *index_regex=None*)

Initalize Wrapper object for standard mzML files.

Parameters

- **path** (*str*) – path to the file
- **encoding** (*str*) – encoding of the file

_binary_search(*target_index*)

Retrieve spectrum for a given spectrum ID using binary jumps

Parameters
target_index (*int*) – native id of the spectrum to access

Returns
pymzML spectrum

Return type
Spectrum (*pymzml.spec.Spectrum*)

_build_index(*from_scratch=False*)

Build an index.

A list of offsets to which a file pointer can seek directly to access a particular spectrum or chromatogram without parsing the entire file.

Parameters
from_scratch (*bool*) – Whether or not to force building the index from scratch, by parsing the file, if no existing index can be found.

Returns
A file-like object used to access the indexed content by seeking to a particular offset for the file.

_build_index_from_scratch(*seeker*)

Build an index of spectra/chromatogram data with offsets by parsing the file.

_interpol_search(*target_index*, *chunk_size=8*, *fallback_cutoff=100*)

Use linear interpolation search to find spectra faster.

Parameters
target_index (*str or int*) – native id of the item to access

Keyword Arguments
chunk_size (*int*) – size of the chunk to read in one go in kb

_read_extremes()

Read min and max spectrum ids. Required for binary jumps.

Returns
list of tuples containing spec_id and file_offset

Return type
seek_list (list)

_search_linear(*seeker*, *index*, *chunk_size=8*)

Fallback to linear search if interpolated search fails.

read(*size=-1*)

Read binary data from file handler.

Keyword Arguments
size (*int*) – Number of bytes to read from file, -1 to read to end of file

Returns
byte string of len size of input data

Return type
data (str)

Gzip

class pymzml.file_classes.standardGzip.**StandardGzip**(*path*, *encoding*)

__getitem__(*identifier*)

Access the item with id 'identifier' in the file by iterating the xml-tree.

Parameters
identifier (*str*) – native id of the item to access

Returns
text associated with the given identifier

Return type
data (str)

__init__(*path*, *encoding*)

Initalize Wrapper object for gzipped mzML files.

Parameters

- **path** (*str*) – path to the file
- **encoding** (*str*) – encoding of the file

_build_index()

Cant build index for standard gzip files

read(*size=-1*)

Read binary data from file handler.

Keyword Arguments
size (*int*) – Number of bytes to read from file, -1 to read to end of file

Returns
byte string of len size of input data

Return type
data (str)

iGzip

class pymzml.file_classes.indexedGzip.**IndexedGzip**(*path*, *encoding*)

__getitem__(*identifier*)

Access the item with id 'identifier' in the file.

Parameters
identifier (*str*) – native id of the item to access

Returns
text associated with the given identifier

Return type
data (str)

__init__(*path*, *encoding*)

Initialize Wrapper object for indexed gzipped files.

Parameters

- **path** (*str*) – path to the file
- **encoding** (*str*) – encoding of the file

_build_index()

Use the GSGR class to retrieve the index from the file and save it.

read(*size=-1*)

Read binary data from file handler.

Keyword Arguments
size (*int*) – Number of bytes to read from file, -1 to read to end of file

Returns
byte string of len size of input data

Return type
data (str)

1.3.6 OBO parser

Class to parse the obo file and set up the accessions library

The OBO parser has been designed to convert MS:xxxxx tags to their appropriate names. A minimal set of MS accession is used in pymzML, but additional accessions can easily be queried.

The obo translator is used internally to associate names with MS:xxxxxxx tags.

The OboTranslator Class generates a dictionary and several lookup tables. e.g.

```
>>> from pymzml.obo import OboTranslator as OT
>>> translator = OT()
>>> translator['MS:1000127']
'centroid mass spectrum'
>>> translator['positive scan']
{'is_a': 'MS:1000465 ! scan polarity', 'id': 'MS:1000130', 'def':
'"Polarity of the scan is positive." [PSI:MS]', 'name': 'positive scan'}
>>> translator['scan']
{'relationship': 'part_of MS:0000000 ! Proteomics Standards Initiative Mass
Spectrometry Ontology', 'id': 'MS:1000441', 'def': '"Function or process of
the mass spectrometer where it records a spectrum." [PSI:MS]', 'name':
'scan'}
>>> translator['unit']
{'relationship': 'part_of MS:0000000 ! Proteomics Standards Initiative Mass
Spectrometry Ontology', 'id': 'MS:1000460', 'def': '"Terms to describe
units." [PSI:MS]', 'name': 'unit'}
```

pymzML comes with the queryOBO.py script that can be used to interrogate the OBO file. Please refer to *Example Scripts* for further usage information.

```
$ ./example_scripts/queryOBO.py "scan time"
MS:1000016
scan time
"The time taken for an acquisition by scanning analyzers." [PSI:MS]
Is a: MS:1000503 ! scan attribute
$
```

```
$ ./example_scripts/queryOBO.py 1000016
MS:1000016
scan time
"The time taken for an acquisition by scanning analyzers." [PSI:MS]
MS:1000503 ! scan attribute
$
```

Accessing specific OBO MS tags

This section describes how to access some common MS tags by their names as they are defined in the OBO file.

First pymzML is imported and the run is defined.

```
>>> example_file = get_example_file.open_example('dta_example.mzML')
>>> import pymzml
>>> msrun = pymzml.run.Reader(example_file)
```

Now, we can fetch specific imformations from the spectrum object.

MS level:

```
>>> for spectrum in msrun:
...     print(spectrum['ms level'])
```

Total Ion current:

```
>>> for spectrum in msrun:
...     print(spectrum['total ion current'])
```

Furthermore we can also check for presence of parameters, therefore the proprties of the spectrum.

Differentiation of e.g. HCD and CID fractionation:

```
>>> for spectrum in msrun:
...     if spctrum['ms level'] == 2:
...         if 'collision-induced dissociation' in spectrum.keys():
...             print('Spectrum {0} is a CID spectrum'.format(spectrum['id']))
...         elif 'high-energy collision-induced dissociation' in spectrum.keys():
...             print('Spectrum {0} is a HCD spectrum'.format(spectrum['id']))
```

1.3.7 Regex patterns

This file hosts a centralized point for all regex patterns used in pymzML

Note: We need some comment lines for each regex, i.e. what does it catch .. maybe with examples and stuff ...

Collection of regular expressions to catch spectrum XML-tags.

pymzml.regex_patterns.**CHROMATOGRAM_AND_SPECTRUM_PATTERN_WITH_ID = re.compile('<\\s*(chromatogram|spectrum)\\s*(id=(\\".*?\\")|index=\\".*?\\")\\s(id=(\\".*?\\"))*\\s*.*\\sdefaultArrayLength=\\"[0-9]+\\">')**

Regex to catch combined chromatogram and spectrum patterns

pymzml.regex_patterns.**CHROMATOGRAM_CLOSE_PATTERN = re.compile(b'</chromatogram>')**

Regex to catch spectrum xml close tags

pymzml.regex_patterns.**CHROMATOGRAM_ID_PATTERN = re.compile('<chromatogram.*id="(.*?)".*?>')**

Regex to catch chromatogram id patterns

pymzml.regex_patterns.**CHROMATOGRAM_PATTERN = re.compile('<chromatogram.*id="(.*?)".*?>')**

Regex to catch chromatogram id pattern (again ?)

pymzml.regex_patterns.**FILE_ENCODING_PATTERN = re.compile(b'encoding="(?P<encoding>[A-Za-z0-9-]*)"')**

Regex to catch xml file encoding

pymzml.regex_patterns.**MOBY_DICK_CHAPTER_PATTERN = re.compile('CHAPTER ([0-9]+).*')**

Regex to catch moby dick chapter number used in the index gezip writer example.

pymzml.regex_patterns.**SIM_INDEX_PATTERN = re.compile(b'(?P<type>idRef=")(?P<nativeID>.*)">(?P<offset>[0-9]*)</offset>')**

Regex pattern for SIM index

pymzml.regex_patterns.**SPECTRUM_CLOSE_PATTERN = re.compile(b'</spectrum>')**

Regex to catch spectrum xml close tags

pymzml.regex_patterns.**SPECTRUM_ID_PATTERN2 = re.compile('(scan|scanId)=(\\d+)')**

Simplified spectrum id regex. Greedly catches ints at the end of line

pymzml.regex_patterns.**SPECTRUM_INDEX_PATTERN = re.compile(b'(?P<type>(scan=|nativeID="))(?P<nativeID>[0-9]*)">"(?P<offset>[0-9]*)</offset>')**

Regex pattern for spectrum index works for obo format 1.1.0 until <last version checked>

Catches:

1. demo 1
2. demo 2

pymzml.regex_patterns.**SPECTRUM_OPEN_PATTERN = re.compile(b'<*spectrum[^>]*(index|id)="(.*?)".*(index|id)="(.*?)"')**

Regex to catch specturm open xml tag with encoded array length

pymzml.regex_patterns.**SPECTRUM_TAG_PATTERN = re.compile('<spectrum.*?id="(?P<index>[^"]+)".*?>')**

Regex to catch spectrum tag pattern

1.4 Examples and Advanced Usage

1.4.1 Writing and Reading igzip files

One key feature of pymzML is the ability to write and read indexed gzip files. The utils module contains a script to easily convert plain or gzipped mzML files into indexed gzip files. However, the *GSGW()* and *GSGR()* class can be used to index any type of data. After these two classes have been introduced, a small example of how to use them for other data than mzML can be found.

1.4.2 Generalized Seekable index Gzip Writer

Please refer to *GSGW()* for further information.

1.4.3 Generalized Seekable index Gzip Reader

Please refer to *GSGR()* for further information.

1.4.4 Practical Example: Moby Dick

Creating the compressed file

To utilze *GSGW()* for other data, one simply needs to parse the data blockwise, so every piece of data, which should be accessible by indexing is written in one go. The index used can be either an integer or a string. The code example below parses each chapter of moby dick and indexes it with its corresponding chapter number.

Example:

```
def index_by_chapter(txt):
    """
    Iterate the text file while collecting the data for each chapter and compressing it

    Args:
        txt(str): Moby Dick text as string
    """
    chapter_start = regex_patterns.MOBY_DICK_CHAPTER_PATTERN
    general_seekable_gzip_writer = GSGW(
        file        = 'Moby_Dick_indexed.gz',
        max_idx     = 50,
        max_idx_len = 2,
        output_path = './Moby_Dick_indexed.gz'
    )

    with general_seekable_gzip_writer as index_writer:
        current_chapter = ''
        for line in txt:
            if re.match(chapter_start, line):
                match = re.match(chapter_start, line)
                current_chapter_number = int(match.group(1)) = 1
                print(
                    'indexing chapter {0} '.format(
                        current_chapter_number
                    ),
                    end = '\r'
                )
                index_writer.add_data(current_chapter, current_chapter_number)
                current_chapter = ''
            else:
                current_chapter += line
        current_chapter_number  += 1
        index_writer.add_data(current_chapter, current_chapter_number)
        print(
```

(continues on next page)

(continued from previous page)

```
            'index chapter {0} '.format(
                current_chapter_number
            ),
            end='\r'
        )
        index_writer.write_index()
        print('Indexed {0} chapters'.format(current_chapter_number))
```

Accessing the data

In order to access the chapter in the compressed file, one simply needs to initialize the *GSGR()* with the path to the created file and can access the chapters conveniently by the python bracket notation ([]).

```
from GSGR import GSGR
import sys
import time

my_Reader = GSGR('./Moby_Dick_indexed.gz')

if __name__ == '__main__':
    if len(sys.argv) != 2:
        print(__doc__)
    else:
        chap_num = int(sys.argv[1])
        print(
            '''
            Reading indexed gzip and retrieving chapter {0}
            '''.format(chap_num)
        )
        s = time.time()
        print(
            '''
            Chapter {0} Took {1:.5f} seconds to retrieve chapter
            '''.format(
                my_Reader.read_block(chap_num),
                time.time() - s
            )
        )
```

1.4.5 igzip file format specification

In the following, the changes from igzip to gzip will be discussed. If one fieldentry requires more than 1 byte, the byte count in indicated in brackets (capital x for arbitrary byte count)

For further information on the gzip format, see https://tools.ietf.org/html/rfc1952 .

ID1	ID2	CM	FLG	MTIME (x4)	XFL	OS

COMPRESSED BLOCKS

CRC32 (x4)	ISIZE (x4)

by setting the 'Comment Flag' in FLG, an additional headerfield can be activated.

file comment, zero-terminated (xX)

This field is then used to save the Uniq IDs, version, index/offset length and is terminated with a zero byte, like described in the following:

ID1	ID2	VERSION	IDXLEN	OFFSETLEN

Index (xX)	Offset (xX)

...

x00

The length of the ID and the offset field can be set when initializing the gzip writer, along with the maximal number of ID/offset pairs.

Example:

```
00000000:  1f8b 0810 ea57 4f5a 0203 4655 0109 06ac 4368 6170 7465 7230  :.....WOZ..FU....
↪Chapter0
00000018:  acac ac36 3436 ac43 6861 7074 6572 31ac 3136 3033 32ac 4368  :...646.Chapter1.
↪16032.Ch
00000030:  6170 7465 7232 ac32 3137 3831 ac43 6861 7074 6572 33ac 3235  :apter2.21781.
↪Chapter3.25
00000048:  3534 37ac 4368 6170 7465 7234 ac33 3932 3436 ac43 6861 7074  :547.Chapter4.
↪39246.Chapt
00000060:  6572 35ac 3433 3435 38ac 4368 6170 7465 7236 ac34 3535 3437  :er5.43458.
↪Chapter6.45547
00000078:  ac43 6861 7074 6572 37ac 3437 3936 39ac 4368 6170 7465 7238  :.Chapter7.47969.
↪Chapter8
00000090:  ac35 3037 3033 ac43 6861 7074 6572 39ac 3533 3239 3943 6861  :.50703.Chapter9.
↪53299Cha
000000a8:  7074 6572 3130 ac36 3230 3138 4368 6170 7465 7231 31ac 3636  :pter10.
↪62018Chapter11.66
000000c0:  3032 3843 6861 7074 6572 3132 ac36 3739 3536 4368 6170 7465  :028Chapter12.
↪67956Chapte
000000d8:  7231 33ac 3730 3333 3043 6861 7074 6572 3134 ac37 3438 3331  :r13.
↪70330Chapter14.74831
000000f0:  4368 6170 7465 7231 35ac 3736 3938 3443 6861 7074 6572 3136  :Chapter15.
↪76984Chapter16
00000108:  ac38 3030 3937 4368 6170 7465 7231 37ac 3933 3134 3743 6861  :.80097Chapter17.
↪93147Cha
00000120:  7074 6572 3138 ac39 3838 3534 4368 6170 7465 7231 3931 3032  :pter18.
↪98854Chapter19102
00000138:  3430 3343 6861 7074 6572 3230 3130 3533 3932 4368 6170 7465 ␣
↪:403Chapter20105392Chapte
```

(continues on next page)

(continued from previous page)

```
00000150:  7232 3131 3037 3739 3043 6861 7074 6572 3232 3131 3037 3232 ␣
↪:r21107790Chapter22110722
00000168:  4368 6170 7465 7232 3331 3134 3936 3143 6861 7074 6572 3234 ␣
↪:Chapter23114961Chapter24
00000180:  3131 3631 3134 4368 6170 7465 7232 3531 3230 3731 3943 6861 ␣
↪:116114Chapter25120719Cha
00000198:  7074 6572 3236 3132 3135 3633 4368 6170 7465 7232 3731 3234 ␣
↪:pter26121563Chapter27124
000001b0:  3839 3343 6861 7074 6572 3238 3132 3933 3138 4368 6170 7465 ␣
↪:893Chapter28129318Chapte
000001c8:  7232 3931 3333 3134 3843 6861 7074 6572 3330 3133 3633 3436 ␣
↪:r29133148Chapter30136346
000001e0:  4368 6170 7465 7233 3131 3337 3230 3043 6861 7074 6572 3332 ␣
↪:Chapter31137200Chapter32
000001f8:  3133 3933 3137 4368 6170 7465 7233 3331 3532 3031 3743 6861 ␣
↪:139317Chapter33152017Cha
00000210:  7074 6572 3334 3135 3437 3635 4368 6170 7465 7233 3531 3630 ␣
↪:pter34154765Chapter35160
00000228:  3536 3743 6861 7074 6572 3336 3136 3732 3736 4368 6170 7465 ␣
↪:567Chapter36167276Chapte
00000240:  7233 3731 3734 3530 3243 6861 7074 6572 3338 3137 3630 3330 ␣
↪:r37174502Chapter38176030
00000258:  4368 6170 7465 7233 3931 3737 3230 3043 6861 7074 6572 3430 ␣
↪:Chapter39177200Chapter40
00000270:  3137 3830 3338 4368 6170 7465 7234 3131 3832 3531 3900
```

In this example of the Moby Dick igz header, the first 10 bytes show the gzip header explained above. After the first 10 bytes, the comment field starts with the 2 ID bytes F and U and version 1. The Idx len is set to have a length of 9 and the offset needs to fit in 6 bytes. After this, the index to offset mapping starts until the zero byte is reached.

1.4.6 Implementing an own file class

In order to make pymzML accept other kinds of mzML data (e.g databases), one can implement an own wrapper similiar to the ones discussed before. In the following, an example for building and accessing a SQL database containing single spectra will be shown.

Creating the wrapper

At first, a database with a specific layout needs to be created. Here, we use a single mzML file and store each spectrum in a table with 2 columns, one for the identifier and one for the xml element of the spectrum in form of a string.

Database creation:

```
import sqlite3
import os
from pymzml import spec
from pymzml.run import Reader

def create_database_from_file(db_name, mzml_path):
    conn = sqlite3.connect(db_name+'.db')
    Run = Reader(os.path.abspath(mzml_path))
```

(continues on next page)

(continued from previous page)

```
    with conn:
        cursor = conn.cursor()
        cursor.execute("CREATE TABLE Spectra(ID INT, xml TEXT)")
        for spec in Run:
            params = (spec.ID, spec.to_string())
            cursor.execute("INSERT INTO Spectra VALUES(?, ?)", params)
    return True
```

After this, we need to implement a class, which needs to implement the __getitem__ function for random access, and a read function used to sequentiallly read in data for iterating the database. In this simple approach, the read function always returns a whole spectra xml string. One obvious optimization would be, to read in smaller chunks of a spec string and jump to the next spectrum, as soon as the end of the current spectrum is reached (as exercise for the interested reader ;)) .

Wrapper for accessing the database:

```
import sqlite3
import os
from pymzml import spec
import xml.etree.ElementTree as et
from pymzml.run import Reader

class SQLiteDatabase(object):
    """
    Example implementation of a database Connector,
    which can be used to make run accept paths to
    sqlite db files.

    We initialize with a path to a database and implement
    a custom __getitem__ function to retrieve the spectra
    """
    def __init__(self, path):
        """
        """
        connection = sqlite3.connect(path)
        self.cursor = connection.cursor()
        self.curr_spec_id = 0

    def __getitem__(self, key):
        """
        """
        self.cursor.execute('SELECT * FROM spectra WHERE id=?', key)
        ID, element = self.cursor.fetchone()

        element = et.XML(element)
        if 'spectrum' in element.tag:
            spectrum = spec.Spectrum(element)
        elif 'chromatogram' in element.tag:
            spectrum = spec.Chromatogram(element)
        return spectrum

    def get_spectrum_count(self):
        self.cursor.execute("SELECT COUNT(*) from spectra")
```

(continues on next page)

(continued from previous page)

```
        num = self.cursor.fetchone()[0]
        return num

    def read(self, size=-1):
        # implement read so it starts reading in first ID,
        # if end reached switches to next id and so on ...
        key = self.current_spectrum_id
        self.cursor.execute('SELECT * FROM spectra WHERE id=?', key)
        ID, element = self.cursor.fetchone()[0]
        self.current_spectrum_id += 1
        return element

if __name__ == '__main__':
    # This is what the Reader class does
    my_iter = iter(et.iterparse(SQLiteDatabase('test.db')))
    # Now you can iter your database
    for x in my_iter:
        print(x)

    # Retrieve a specific spectrum from your database
    db = SQLiteDatabase('test.db')
    unique_id = 5
    my_spec = db[unique_id]
```

Enabling the wrapper

In order to allow pymzML to use this new file class, the filehandler needs to be able to detect when to use this class. The easiest way is, to add another elif statement which decides which handler to use based on the file path. For this, edit the *_open()* method as shown in the following:

Code:

```
def _open(self, path):
    """
    Open a file like object resp. a wrapper for a file like object.

    Arguments:
        path (str): path to the mzml file

    Returns:
        file_handler: instance of
        :py:class:`~pymzml.file_classes.standardGzip.StandardGzip`,
        :py:class:`~pymzml.file_classes.indexedGzip.IndexedGzip` or
        :py:class:`~pymzml.file_classes.standardMzml.StandardMzml`,
        based on the file ending of 'path'
    """
    if path.endswith('.gz'):
        if self._indexed_gzip(path):
            # set offset_names and self.offsets
            self.file_handler = indexedGzip.IndexedGzip(path, self.encoding)
            # for k, v in self.file_handler.index.items():
```

(continues on next page)

(continued from previous page)

```
            #     self.offset_names.append( k )
            #     self.offsets.append( v )
            # self.offset_names   = [key for key in ra_reader.index.keys()]
            # self.offsets        = [off for off in ra_reader.index.values()]
            #, self.as_numpy
        else:
            self.file_handler = standardGzip.StandardGzip(path, self.encoding)
    # add your new elif statement here
    elif path.endswith('db'):
        from SQLiteConnector import SQLiteDatabase
        self.file_handler = SQLiteDatabase(path, encoding)
    else:
        self.file_handler     = standardMzml.StandardMzml(path, self.encoding)
    return self.file_handler
```

1.4.7 Example Scripts

Parsing a mzML file (new syntax)

simple_parser.**main**(*mzml_file*)

Basic example script to demonstrate the usage of pymzML. Requires a mzML file as first argument.

usage:

./simple_parser.py <path_to_mzml_file>

Note: This script uses the new syntax with the MS level being a property of the spectrum class (Spectrum.ms_level). The old syntax can be found in the script simple_parser_v2.py where the MS level can be queried as a key (Spectrum['ms level'])

```
#!/usr/bin/env python
import sys
import pymzml


def main(mzml_file):
    """
    Basic example script to demonstrate the usage of pymzML. Requires a mzML
    file as first argument.

    usage:

        ./simple_parser.py <path_to_mzml_file>

    Note:

        This script uses the new syntax with the MS level being a property of
        the spectrum class ( Spectrum.ms_level ). The old syntax can be found in
        the script simple_parser_v2.py where the MS level can be queried as a key
        (Spectrum['ms level'])
```

(continues on next page)

(continued from previous page)

```
    """
    run = pymzml.run.Reader(mzml_file)
    for n, spec in enumerate(run):
        print(
            "Spectrum {0}, MS level {ms_level} @ RT {scan_time:1.2f}".format(
                spec.ID, ms_level=spec.ms_level, scan_time=spec.scan_time_in_minutes()
            )
        )
    print("Parsed {0} spectra from file {1}".format(n, mzml_file))
    print()


if __name__ == "__main__":
    if len(sys.argv) < 2:
        print(main.__doc__)
        exit()
    mzml_file = sys.argv[1]
    main(mzml_file)
```

Parsing a mzML file (old syntax)

simple_parser_v2.**main**(*mzml_file*)

Basic example script to demonstrate the usage of pymzML. Requires a mzML file as first argument.

usage:

./simple_parser_v2.py <path_to_mzml_file>

Note: This script uses the old syntax where the MS level can be queried as a key (Spectrum['ms level']). The current syntax can be found in simple_parser.py

---

```
#!/usr/bin/env python
import sys
import pymzml
from collections import defaultdict as ddict


def main(mzml_file):
    """
    Basic example script to demonstrate the usage of pymzML. Requires a mzML
    file as first argument.

    usage:
        ./simple_parser_v2.py <path_to_mzml_file>

    Note:

        This script uses the old syntax where the MS level can be queried as a
        key (Spectrum['ms level']). The current syntax can be found in
```

(continues on next page)

(continued from previous page)

```
        simple_parser.py

    """
    run = pymzml.run.Reader(mzml_file)
    # print( run[10000].keys() )
    stats = ddict(int)
    for n, spec in enumerate(run):
        print(
            "Spectrum {0}, MS level {ms_level}".format(n, ms_level=spec["ms level"]),
            end="\r",
        )
        # the old method to obtain peaks from the Spectrum class
        stats[spec.ID] = len(spec.centroidedPeaks)

    print("Parsed {0} spectra from file {1}".format(len(stats.keys()), mzml_file))
    print()


if __name__ == "__main__":
    if len(sys.argv) < 2:
        print(main.__doc__)
        exit()
    mzml_file = sys.argv[1]
    main(mzml_file)
```

Query the obo files

queryOBO.**main**(*args*)

Use this script to interrogate the OBO database files.

usage:

./queryOBO.py [-h] [-v VERSION] query

Example:

```
$ ./queryOBO.py'scan time'
MS:1000016
scan time
'The time taken for an acquisition by scanning analyzers.' [PSI:MS]
Is a: MS:1000503 ! scan attribute
```

Example:

```
$ ./queryOBO.py 1000016
MS:1000016
scan time
"The time taken for an acquisition by scanning analyzers." [PSI:MS]
MS:1000503 ! scan attribute
```

```
#!/usr/bin/env python
# -*- coding: utf-8 -*-
```

(continues on next page)

(continued from previous page)

```
import argparse
from collections import defaultdict

import pymzml.obo


FIELDNAMES = ["id", "name", "def", "is_a"]


def main(args):
    """
    Use this script to interrogate the OBO database files.

    usage:

        ./queryOBO.py [-h] [-v VERSION] query

    Example::

      $ ./queryOBO.py'scan time'
      MS:1000016
      scan time
      'The time taken for an acquisition by scanning analyzers.' [PSI:MS]
      Is a: MS:1000503 ! scan attribute

    Example::

        $ ./queryOBO.py 1000016
        MS:1000016
        scan time
        "The time taken for an acquisition by scanning analyzers." [PSI:MS]
        MS:1000503 ! scan attribute


    """
    obo = pymzml.obo.OboTranslator(version=args.version)
    obo.parseOBO()
    if args.query.isdigit():
        print(search_by_id(obo, args.query))
    else:
        for ix, match in enumerate(search_by_name(obo, args.query)):
            print("#{0}".format(ix))

            for fieldname in ("id", "name", "def"):
                print(match[fieldname])

            if "is_a" in match:
                print("Is a:", match["is_a"])
```

(continues on next page)

(continued from previous page)

```
def search_by_name(obo, name):
    print("Searching for {0}".format(name.lower()))
    matches = []
    for lookup in obo.lookups:
        for key in lookup.keys():
            if name.lower() in key.lower():
                match = defaultdict(str)

                for fieldname in FIELDNAMES:
                    if fieldname in lookup[key].keys():
                        match[fieldname] = lookup[key][fieldname]

                matches.append(match)

    return matches


def search_by_id(obo, id):
    key = "MS:{0}".format(id)
    return_value = ""
    for lookup in obo.lookups:
        if key in lookup:
            if obo.MS_tag_regex.match(key):
                for fn in FIELDNAMES:
                    if fn in lookup[key].keys():
                        return_value += "{0}\n".format(lookup[key][fn])
    return return_value


if __name__ == "__main__":
    argparser = argparse.ArgumentParser(usage=__doc__)
    argparser.add_argument(
        "query", help="an accession or part of an OBO term name to look for"
    )
    argparser.add_argument(
        "-v",
        "--version",
        default="1.1.0",
        help="""
            the version of the OBO to use; valid options are 1.0.0, 1.1.0, and 1.2,
            default is 1.1.0
        """,
    )

    args = argparser.parse_args()

    main(args)
```

Plotting a chromatogram

plot_chromatogram.**main**(*mzml_file*)

Plots a chromatogram for the given mzML file. File is saved as 'chromatogram_<mzml_file>.html'.

usage:

./plot_chromatogram.py <path_to_mzml_file>

```
#!/usr/bin/env python
# -*- coding: utf-8 -*-

import os
import sys

import pymzml
from pymzml.plot import Factory


def main(mzml_file):
    """
    Plots a chromatogram for the given mzML file. File is saved as
    'chromatogram_<mzml_file>.html'.

    usage:

        ./plot_chromatogram.py <path_to_mzml_file>

    """
    run = pymzml.run.Reader(mzml_file)
    mzml_basename = os.path.basename(mzml_file)
    pf = Factory()
    pf.new_plot()
    pf.add(run["TIC"].peaks(), color=(0, 0, 0), style="lines", title=mzml_basename)
    pf.save(
        "chromatogram_{0}.html".format(mzml_basename),
        layout={
            "xaxis":{
                "title": "Retention time",
                "ticks": 'outside',
                "ticklen": 2,
                "tickwidth": 0.25,
                "showgrid": False,
                "linecolor": 'black',
            },
            "yaxis": {
                "title": "TIC",
                "ticks": 'outside',
                "ticklen": 2,
                "tickwidth": 0.25,
                "showgrid": False,
                "linecolor": 'black',
            },
            "plot_bgcolor": 'rgba(255, 255, 255, 0)',
```

(continues on next page)

(continued from previous page)

```
            "paper_bgcolor": 'rgba(255, 255, 255, 0)',
        },
    )
    return


if __name__ == "__main__":
    if len(sys.argv) < 2:
        print(main.__doc__)
        exit()
    mzml_file = sys.argv[1]
    main(mzml_file)
```

Plotting a spectrum

plot_spectrum.**main**()

This function shows how to plot a simple spectrum. It can be directly plotted via this script or using the python console.

usage:

./plot_spectrum.py

```
#!/usr/bin/env python
# -*- coding: utf-8 -*-

import os

import pymzml


def main():
    """
    This function shows how to plot a simple spectrum. It can be directly
    plotted via this script or using the python console.

    usage:

        ./plot_spectrum.py

    """

    example_file = os.path.join(
        os.path.dirname(__file__), os.pardir, "tests", "data", "example.mzML"
    )
    run = pymzml.run.Reader(example_file)
    p = pymzml.plot.Factory()
    for spec in run:
        p.new_plot()
        p.add(spec.peaks("centroided"), color=(0, 0, 0), style="sticks", name="peaks")
        filename = "example_plot_{0}_{1}.html".format(
            os.path.basename(example_file), spec.ID
```

(continues on next page)

(continued from previous page)

```
        )
        p.save(
            filename=filename,
            layout={
                "xaxis":{
                    "ticks": 'outside',
                    "ticklen": 2,
                    "tickwidth": 0.25,
                    "showgrid": False,
                    "linecolor": 'black',
                },
                "yaxis": {
                    "ticks": 'outside',
                    "ticklen": 2,
                    "tickwidth": 0.25,
                    "showgrid": False,
                    "linecolor": 'black',
                },
                "plot_bgcolor": 'rgba(255, 255, 255, 0)',
                "paper_bgcolor": 'rgba(255, 255, 255, 0)',
            },
        )
        print("Plotted file: {0}".format(filename))
        break


if __name__ == "__main__":
    main()
```

Plotting a spectrum with annotation

plot_spectrum_with_annotation.**main**()

This script shows how to plot multiple spectra in one plot and how to use label for the annotation of spectra. The first plot is an MS1 spectrum with the annotated precursor ion. The second plot is a zoom into the precursor isotope pattern. The third plot is an annotated fragmentation spectrum (MS2) of the peptide HLVDEPQNLIK from BSA. These examples also show the use of 'layout' to define the appearance of a plot.

usage:

./plot_spectrum_with_annotation.py

```
#!/usr/bin/env python
# -*- coding: utf-8 -*-

import copy
import os

import pymzml


def main():
    """
```

(continues on next page)

(continued from previous page)

```
    This script shows how to plot multiple spectra in one plot and
    how to use label for the annotation of spectra.
    The first plot is an MS1 spectrum with the annotated precursor ion.
    The second plot is a zoom into the precursor isotope pattern.
    The third plot is an annotated fragmentation spectrum (MS2) of the
    peptide HLVDEPQNLIK from BSA.
    These examples also show the use of 'layout' to define the appearance
    of a plot.

    usage:

        ./plot_spectrum_with_annotation.py

    """

    # First we define some general layout attributes
    layout = {
        "xaxis": {
            "title": "<i>m/z</i>",
            "ticklen": 5,
            "tickwidth": 1,
            "ticks": "outside",
            "showgrid": False,
            "linecolor": 'black',
        },
        "yaxis": {
            "ticklen": 5,
            "tickwidth": 1,
            "ticks": "outside",
            "showgrid": False,
            "linecolor": 'black',
        },
    }

    # The example BSA file will be used
    example_file = os.path.join(
        os.path.dirname(__file__), os.pardir, "tests", "data", "BSA1.mzML.gz"
    )

    # Define different precisions for MS1 and MS2
    run = pymzml.run.Reader(example_file, MS_precisions={1: 5e-6, 2: 5e-4})
    p = pymzml.plot.Factory()
    plot_layout = {}

    # Now that everything is set up, we can plot the MS1 spectrum
    # Spectrum ID: 1574
    p.new_plot(title="MS1 Spectrum")
    ms1_spectrum = run[1574]

    # The measured peaks are added as first trace
    p.add(
        ms1_spectrum.peaks("centroided"),
```

(continues on next page)

(continued from previous page)

```
        color=(0, 0, 0),
        opacity=1,
        style="sticks",
        name="raw data plot 1",
    )

    # The label for the precursor ion is added as a seperate trace.
    # Note that triangle.MS_precision is used here as a label.
    # By zooming in at this peak one can therefore check if the measured
    # peak fits into defined the mass accuracy range.
    precursor_mz_calc = 435.9102
    p.add(
        [(precursor_mz_calc, "max_intensity", "theoretical precursor")],
        color=(255, 0, 0),
        opacity=0.6,
        style="label.triangle.MS_precision",
        name="theoretical precursor plot 1",
    )

    # Define the layout for the first subplot.
    # The x- and y-axes of subplots are numbered, starting at 1.
    for axis in layout.keys():
        plot_layout["{0}1".format(axis)] = copy.copy(layout[axis])

    # Now we can add a second plot, the same way as above but as a zoom-in.
    # Therefore, we define a mz_range
    p.new_plot(title="MS1 Spectrum Zoom")
    p.add(
        ms1_spectrum.peaks("centroided"),
        color=(0, 0, 0),
        opacity=1,
        style="sticks",
        name="raw data plot 2",
        plot_num=1,
        mz_range=[435.7, 437],
    )

    p.add(
        [(precursor_mz_calc, "max_intensity", "theoretical precursor")],
        color=(255, 0, 0),
        opacity=0.3,
        plot_num=1,
        style="label.triangle.MS_precision",
        name="theoretical precursor plot 2",
    )

    # The mz_range can be included in the layout as well.
    # In contrast to mz_range in the add() function, which limits the included
    # datapoints, the layout range only defines the area that is depicted (i.e. the zoom)
    for axis in layout.keys():
        plot_layout["{0}2".format(axis)] = copy.copy(layout[axis])
    plot_layout["xaxis2"]["autorange"] = False
```

(continues on next page)

(continued from previous page)

```
    plot_layout["xaxis2"]["range"] = [435.7, 437]

    # Now the third plot will be added, a fragmentation spectrum of HLVDEPQNLIK
    ms2_spectrum = run[3542]

    # The MS_precision for the plotting option label.triangle.MS_precision
    # needs to be defined
    p.new_plot(title="MS2 Spectrum Annotated: HLVDEPQNLIK", MS_precision=5e-4)
    p.add(
        ms2_spectrum.peaks("centroided"),
        color=(0, 0, 0),
        opacity=1,
        style="sticks",
        name="raw data plot 3",
        plot_num=2,
    )

    theoretical_b_ions = {
        "b<sub>2</sub><sup>+2</sup>": 126.0788,
        "b<sub>3</sub><sup>+2</sup>": 175.6130,
        "b<sub>4</sub><sup>+2</sup>": 233.1264,
        "b<sub>2</sub>": 251.1503,
        "b<sub>5</sub><sup>+2</sup>": 297.6477,
        "b<sub>6</sub><sup>+2</sup>": 346.1741,
        "b<sub>3</sub>": 350.2187,
        "b<sub>7</sub><sup>+2</sup>": 410.2034,
        "b<sub>4</sub>": 465.2456,
        "b<sub>8</sub><sup>+2</sup>": 467.2249,
        "b<sub>9</sub><sup>+2</sup>": 523.7669,
        "b<sub>10</sub><sup>+2</sup>": 580.3089,
        "b<sub>5</sub>": 594.2882,
        "b<sub>6</sub>": 691.341,
        "b<sub>7</sub>": 819.3995,
        "b<sub>8</sub>": 933.4425,
        "b<sub>9</sub>": 1046.5265,
        "b<sub>10</sub>": 1159.6106,
    }

    theoretical_y_ions = {
        "y<sub>1</sub><sup>+2</sup>": 74.0600,
        "y<sub>2</sub><sup>+2</sup>": 130.6021,
        "y<sub>1</sub>": 147.1128,
        "y<sub>3</sub><sup>+2</sup>": 187.1441,
        "y<sub>4</sub><sup>+2</sup>": 244.1656,
        "y<sub>2</sub>": 260.1969,
        "y<sub>5</sub><sup>+2</sup>": 308.1949,
        "y<sub>6</sub><sup>+2</sup>": 356.7212,
        "y<sub>3</sub>": 373.2809,
        "y<sub>7</sub><sup>+2</sup>": 421.2425,
        "y<sub>8</sub><sup>+2</sup>": 478.7560,
        "y<sub>4</sub>": 487.3239,
        "y<sub>9</sub><sup>+2</sup>": 528.2902,
```

(continues on next page)

(continued from previous page)

```
        "y<sub>10</sub><sup>+2</sup>": 584.8322,
        "y<sub>5</sub>": 615.3824,
        "y<sub>6</sub>": 712.4352,
        "y<sub>7</sub>": 841.4778,
        "y<sub>8</sub>": 956.5047,
        "y<sub>9</sub>": 1055.5732,
        "y<sub>10</sub>": 1168.6572,
    }

    # Check which theoretical fragments are present in the spectrum
    # using the has_peak() function
    for ion_list in [theoretical_b_ions, theoretical_y_ions]:
        label_list = []
        for fragment in ion_list.keys():
            peak = ms2_spectrum.has_peak(ion_list[fragment])
            if len(peak) != 0:
                label_list.append((ion_list[fragment], peak[0][1], fragment))
        if ion_list == theoretical_b_ions:
            color = (0, 0, 255)
        else:
            color = (0, 255, 0)
        p.add(
            label_list,
            color=color,
            style="label.triangle.MS_precision",
            name="theoretical fragment ions plot 3",
        )

    for axis in layout.keys():
        plot_layout["{0}3".format(axis)] = copy.copy(layout[axis])

    plot_layout["plot_bgcolor"] = 'rgba(255, 255, 255, 0)'
    plot_layout["paper_bgcolor"] = 'rgba(255, 255, 255, 0)'

    # Save the plot in a file using the defined plot_layout
    filename = "example_plot_{0}_annotation.html".format(os.path.basename(example_file))
    p.save(filename=filename, layout=plot_layout)
    print("Plotted file: {0}".format(filename))


if __name__ == "__main__":
    main()
```

Extracting highest peaks

highest_peaks.**main**()

Testscript to isolate the n-highest peaks from an example file.

Usage:

./highest_peaks.py

Parses the file '../tests/data/example.mzML' and extracts the 2 highest intensities from each spectrum.

```
#!/usr/bin/env python

import pymzml
from collections import defaultdict as ddict
import os


def main():
    """
    Testscript to isolate the n-highest peaks from an example file.

    Usage:

        ./highest_peaks.py

    Parses the file '../tests/data/example.mzML' and extracts the 2 highest
    intensities from each spectrum.

    """
    example_file = os.path.join(
        os.path.dirname(__file__), os.pardir, "tests", "data", "example.mzML"
    )
    run = pymzml.run.Reader(example_file)
    highest_i_dict = ddict(list)

    number_of_peaks_to_extract = 2

    for spectrum in run:
        # print( spectrum.ID )
        if spectrum.ms_level == 1:
            for mz, i in spectrum.highest_peaks(number_of_peaks_to_extract):
                highest_i_dict[spectrum.ID].append(i)
    for spectrum_id, highest_peak_list in highest_i_dict.items():
        assert len(highest_peak_list) == number_of_peaks_to_extract
        print(
            "Spectrum {0}; highest intensities: {1}".format(
                spectrum_id, highest_peak_list
            )
        )


if __name__ == "__main__":
    main()
```

Compare spectra

compare_spectra.**main**()

Compare multiple spectra and return the cosine distance between them. The returned value is between 0 and 1, a returned value of 1 represents highest similarity.

usage:

./compare_spectra.py

```
#!/usr/bin/env python
# -*- coding: utf-8 -*-

import os

import pymzml


def main():
    """
    Compare multiple spectra and return the cosine distance between them.
    The returned value is between 0 and 1, a returned value of 1
    represents highest similarity.

    usage:

        ./compare_spectra.py

    """
    example_file = os.path.join(
        os.path.dirname(__file__), os.pardir, "tests", "data", "example.mzML"
    )
    print(
        """
            Comparing spectra
        """
    )
    # print(example_file)
    run = pymzml.run.Reader(example_file)
    tmp = []
    for spec in run:
        if spec.ms_level == 1:
            print("Parsing spectrum lvl 1 has id {0}".format(spec.ID))
            tmp.append(spec)
            if len(tmp) >= 3:
                break

    print("Print total number of specs collected {0}".format(len(tmp)))
    for compare_tuples in [(0, 1), (0, 2), (1, 2)]:
        print(
            "Cosine between spectra {0} & {1} is {2:1.4f}".format(
                compare_tuples[0] + 1,
                compare_tuples[1] + 1,
                tmp[compare_tuples[0]].similarity_to(tmp[compare_tuples[1]]),
```

(continues on next page)

(continued from previous page)

```
            )
        )

    print(
        "Cosine score between first spectrum against itself: {0:1.4f}".format(
            tmp[0].similarity_to(tmp[0])
        )
    )


if __name__ == "__main__":
    main()
```

Find m/z values

has_peak.**main**()

Testscript to demonstrate functionality of function *pymzml.spec.Spectrum.has_peak()*

usage:

./has_peak.py

```
#!/usr/bin/env python

import pymzml
import os


def main():
    """
    Testscript to demonstrate functionality of function :py:func:`pymzml.spec.Spectrum.
↪has_peak`

    usage:

        ./has_peak.py

    """

    example_file = os.path.join(
        os.path.dirname(__file__), os.pardir, "tests", "data", "example.mzML"
    )
    mz_to_find = 820.7711792
    run = pymzml.run.Reader(example_file)
    for spectrum in run:
        found_peaks = spectrum.has_peak(mz_to_find)
        if found_peaks != []:
            print("Found peaks: {0} in spectrum {1}".format(found_peaks, spectrum.ID))


if __name__ == "__main__":
    main()
```

Extract ion chromatogram

extract_ion_chromatogram.**main**()

Demonstration of the extraction of a specific ion chromatogram, i.e. XIC or EIC

All intensities and m/z values for a target m/z are extracted.

usage:

./extract_ion_chromatogram.py

```
#!/usr/bin/env python

import os

import pymzml


def main():
    """
    Demonstration of the extraction of a specific ion chromatogram, i.e. XIC or EIC

    All intensities and m/z values for a target m/z are extracted.

    usage:

        ./extract_ion_chromatogram.py

    """

    example_file = os.path.join(
        os.path.dirname(__file__), os.pardir, "tests", "data", "example.mzML"
    )
    run = pymzml.run.Reader(example_file)
    time_dependent_intensities = []

    MZ_2_FOLLOW = 70.06575775

    for spectrum in run:
        if spectrum.ms_level == 1:
            has_peak_matches = spectrum.has_peak(MZ_2_FOLLOW)
            if has_peak_matches != []:
                for mz, I in has_peak_matches:
                    time_dependent_intensities.append(
                        [spectrum.scan_time_in_minutes(), I, mz]
                    )
    print("RT   \ti   \tmz")
    for rt, i, mz in time_dependent_intensities:
        print("{0:5.3f}\t{1:13.4f}\t{2:10}".format(rt, i, mz))
    return


if __name__ == "__main__":
    main()
```

Find abundant precursors

get_precursors.**main**()

Extract the 10 most often fragmented precursors from the BSA example file.

This can e.g. be used for defining exclusion lists for further MS runs.

usage:

./get_precursors.py

```
#!/usr/bin/env python

import os
from operator import itemgetter

import pymzml


def main():
    """
    Extract the 10 most often fragmented precursors from the BSA example file.

    This can e.g. be used for defining exclusion lists for further MS runs.

    usage:

        ./get_precursors.py

    """

    example_file = os.path.join(
        os.path.dirname(__file__), os.pardir, "tests", "data", "BSA1.mzML.gz"
    )
    run = pymzml.run.Reader(example_file)
    fragmented_precursors = {}
    for spectrum in run:
        if spectrum.ms_level == 2:
            selected_precursors = spectrum.selected_precursors
            if spectrum.selected_precursors is not None:
                for precursor_dict in selected_precursors:
                    precursor_mz = precursor_dict["mz"]
                    precursor_i = precursor_dict["i"]
                    rounded_precursor_mz = round(precursor_mz, 3)
                    if rounded_precursor_mz not in fragmented_precursors.keys():
                        fragmented_precursors[rounded_precursor_mz] = []
                    fragmented_precursors[rounded_precursor_mz].append(spectrum.ID)

    precursor_info_list = []
    for rounded_precursor_mz, spectra_list in fragmented_precursors.items():
        precursor_info_list.append(
            (len(spectra_list), rounded_precursor_mz, spectra_list)
        )

    for pos, (number_of_spectra, rounded_precursor_mz, spectra_list) in enumerate(
```

(continues on next page)

(continued from previous page)

```
        sorted(precursor_info_list, reverse=True)
    ):
        print(
            "Found precursor: {0} in spectra: {1}".format(
                rounded_precursor_mz, spectra_list
            )
        )
        if pos > 8:
            break


if __name__ == "__main__":
    main()
```

Access polarity of spectra

polarity.**main**()

Accessing positive or negative polarity of scan using obo 1.1.0

usage:

./polarity.py

```
#!/usr/bin/env python
# -*- coding: utf-8 -*-


import pymzml
import get_example_file


def main():
    """
    Accessing positive or negative polarity of scan using obo 1.1.0

    usage:

        ./polarity.py

    """
    example_file = get_example_file.open_example("small.pwiz.1.1.mzML")
    run = pymzml.run.Reader(example_file, obo_version="1.1.0")
    for spec in run:
        negative_polarity = spec["negative scan"]
        if negative_polarity is None:
            negative_polarity = False
        if negative_polarity == "":
            negative_polarity = True
        positive_polarity = spec["positive scan"]
        if positive_polarity is None:
            positive_polarity = False
        if positive_polarity == "":
```

(continues on next page)

(continued from previous page)

```
            positive_polarity = True
        else:
            positive_polarity = False
        print(
            "Polarity negative {0} - Polarity positive {1}".format(
                negative_polarity, positive_polarity
            )
        )
        exit(1)

    return


if __name__ == "__main__":
    main()
```

Check old to new function name mapping

deprecation_check.**main**()

Testscript to highlight the function name changes in the Spectrum class.

Note: Please adjust any old scripts to the new syntax.

usage:

./deprecation_check.py

```
#!/usr/bin/env python3

import os

import pymzml


def main():
    """
    Testscript to highlight the function name changes in the Spectrum class.

    Note:
        Please adjust any old scripts to the new syntax.

    usage:

        ./deprecation_check.py

    """

    example_file = os.path.join(
        os.path.dirname(__file__), os.pardir, "tests", "data", "example.mzML"
    )
```

(continues on next page)

(continued from previous page)

```
    run = pymzml.run.Reader(example_file)
    spectrum_list = []
    for pos, spectrum in enumerate(run):
        spectrum_list.append(spectrum)
        spectrum.hasPeak((813.19073486))
        spectrum.extremeValues("mz")
        spectrum.hasOverlappingPeak(813.19073486)
        spectrum.highestPeaks(1)
        spectrum.estimatedNoiseLevel()
        spectrum.removeNoise()
        spectrum.transformMZ(813.19073486)
        if pos == 1:
            spectrum.similarityTo(spectrum_list[0])
            break


if __name__ == "__main__":
    main()
```

Convert mzML(.gz) to mzML.gz (igzip)

gzip_mzml.**main**(*mzml_path*, *out_path*)

Create and indexed gzip mzML file from a plain mzML.

Usage: python3 gzip_mzml.py <path/to/mzml> <path/to/output>

```
#!/usr/bin/env python3

import sys
import os
from pymzml.utils.utils import index_gzip
from pymzml.run import Reader


def main(mzml_path, out_path):
    """
    Create and indexed gzip mzML file from a plain mzML.

    Usage: python3 gzip_mzml.py <path/to/mzml> <path/to/output>
    """
    with open(mzml_path) as fin:
        fin.seek(0, 2)
        max_offset_len = fin.tell()
        max_spec_no = Reader(mzml_path).get_spectrum_count() + 10

    index_gzip(
        mzml_path, out_path, max_idx=max_spec_no, idx_len=len(str(max_offset_len))
    )


if __name__ == "__main__":
```

(continues on next page)

(continued from previous page)

```
    if len(sys.argv) > 2:
        main(sys.argv[1], sys.argv[2])
    else:
        print(main.__doc__)
```

Multi threading conversion of mzML(.gz) to mzML.gz (igzip)

multi_threading_file_compression.**main**(*folder*, *num_cpus=1*)

Creates indexed gzip mzML files from all mzMLs files in the given folder using a given number of threads.

Usage:

python multi_threading_file_compression.py <folder> <threads>

Note: If the number of threads is larger than the number of actual possible threads, all possible threads will be used.

```
#!/usr/bin/env python3

import sys
import os
from pymzml.utils.utils import index_gzip
import pymzml
import glob
import multiprocessing


def main(folder, num_cpus=1):
    """
    Creates indexed gzip mzML files from all mzMLs files in the given folder
    using a given number of threads.

    Usage:
        python multi_threading_file_compression.py <folder> <threads>

    Note:
        If the number of threads is larger than the number of actual possible
        threads, all possible threads will be used.

    """
    max_cpus = multiprocessing.cpu_count()
    if int(num_cpus) > max_cpus:
        num_cpus = max_cpus
    else:
        num_cpus = int(num_cpus)
    mzml_job_list = []
    for mzml_path in glob.glob(os.path.join(folder, "*.mzML")):
        out_path = "{0}.gz".format(mzml_path)
        if os.path.exists(out_path):
            print("Skipping: {0}".format(mzml_path))
            continue
```

(continues on next page)

(continued from previous page)

```
        mzml_job_list.append((mzml_path, out_path))
    print(
        "Compressing {0} mzML files using {1} threads".format(
            len(mzml_job_list), num_cpus
        )
    )
    mp_pool = multiprocessing.Pool(num_cpus)
    results = mp_pool.starmap(compress_file, mzml_job_list)
    mp_pool.close()
    print("Done")
    return


def compress_file(file_path, out_path):
    print("Working on file {0}".format(file_path))
    with open(file_path) as fin:
        fin.seek(0, 2)
        max_offset_len = fin.tell()
        max_spec_no = pymzml.run.Reader(file_path).get_spectrum_count() + 10

    index_gzip(
        file_path, out_path, max_idx=max_spec_no, idx_len=len(str(max_offset_len))
    )
    print("Wrote file {0}".format(out_path))
    return


if __name__ == "__main__":
    if len(sys.argv) < 2:
        print(main.__doc__)
        exit()
    else:
        main(*sys.argv[1:])
```

Acces run infos

access_run_info.**main**(*mzml_file*)

Basic example script to access basic run info of an mzML file. Requires a mzML file as first command line argument.

usage:

./access_run_info.py <path_to_mzml_file>

```
>>> run.info =
        {
            'encoding': 'utf-8',
             'file_name': '/Users/joe/Dev/pymzml_2.0/tests/data/BSA1.mzML.gz',
             'file_object': <pymzml.file_interface.FileInterface object at␣
↪0x1039a3f28>,
             'obo_version': '1.1.0',
             'offset_dict': None,
```

(continues on next page)

(continued from previous page)

```
            'run_id': 'ru_0',
            'spectrum_count': 1684,
            'start_time': '2009-08-09T22:32:31'
        }
```

```
#!/usr/bin/env python
import sys
import pymzml


def main(mzml_file):
    """
    Basic example script to access basic run info of an mzML file. Requires a
    mzML file as first command line argument.

    usage:

        ./access_run_info.py <path_to_mzml_file>

    >>> run.info =
            {
                'encoding': 'utf-8',
                'file_name': '/Users/joe/Dev/pymzml_2.0/tests/data/BSA1.mzML.gz',
                'file_object': <pymzml.file_interface.FileInterface object at␣
↪0x1039a3f28>,
                'obo_version': '1.1.0',
                'offset_dict': None,
                'run_id': 'ru_0',
                'spectrum_count': 1684,
                'start_time': '2009-08-09T22:32:31'
             }

    """
    run = pymzml.run.Reader(mzml_file)
    print(
        """
Summary for mzML file:
    {file_name}
Run was measured on {start_time} using obo version {obo_version}
File contains {spectrum_count} spectra
        """.format(
            **run.info
        )
    )


if __name__ == "__main__":
    if len(sys.argv) < 2:
        print(main.__doc__)
        exit()
    mzml_file = sys.argv[1]
    main(mzml_file)
```

Creating a custom Filehandler

Introduction

It is also possible to create an own API for different forms of mzML files. For this, a new class needs to be written, which implements a *read* and a *__getitem__* function.

Implementation of the API Class

Example:

```
class SQLiteDatabase(object):
        """
        Example implementation of a database Conncetor,
        which can be used to make run accept paths to
        sqlite db files.

        We initialize with a path to a database and implement
        a custom __getitem__ function to retrieve the spectra
        """

        def __init__(self, path):
                """
                """
                connection = sqlite3.connect(path)
                self.cursor = connection.cursor()

        def __getitem__(self, key):
                """
                Execute a SQL request, process the data and return a spectrum object.

                Args:
                        key (str or int): unique identifier for the given spectrum in the
                        database
                """
                self.cursor.execute('SELECT * FROM spectra WHERE id=?', key)
                ID, element = self.cursor.fetchone()

                element = et.XML(element)
                if 'spectrum' in element.tag:
                        spectrum = spec.Spectrum(element)
                elif 'chromatogram' in element.tag:
                        spectrum = spec.Chromatogram(element)
                return spectrum

        def get_spectrum_count(self):
                self.cursor.execute("SELECT COUNT(*) from spectra")
                num = self.cursor.fetchone()[0]
                return num

        def read(self, size=-1):
                # implement read so it starts reading in first ID,
```

(continues on next page)

(continued from previous page)

```
                # if end reached switches to next id and so on ...

                return '<spectrum index="0" id="controllerType=0 controllerNumber=1␣
↪scan=1" defaultArrayLength="917"></spectrum>\n'
```

Enabling the new API Class in File Interface

In order to make the run class accept the new file class, one need to edit the `_open()` function in file_interface.py

Example:

```
def _open(self, path):
        if path.endswith('.gz'):
                if self._indexed_gzip(path):
                        self.file_handler = indexedGzip.IndexedGzip(path, self.encoding)
                else:
                        self.file_handler = standardGzip.StandardGzip(path, self.
↪encoding)
        # Insert a new condition to enable your new fileclass
        elif path.endswith('.db'):
                self.file_handler = utils.SQLiteConnector.SQLiteDatabase(path, self.
↪encoding)
        else:
                self.file_handler     = standardMzml.StandardMzml(path, self.encoding)
        return self.file_handler
```

Moby Dick as indexed Gzip

Example of how to use the GSGW and GSGR class to create and access indexed Gzip files

```
python3 index_moby_dick.py

python3 read_moby_dick.py 10
```

1.5 Supplemental Material

1.5.1 Benchmark Files

All samples contain TMT labeled E.Coli cells on human background. The different Instruments (Orbitrap, Lumos, HF, HF-X) were run in DDA mode.

Machines:

HF :

P0109699E18

P0109699E19

P0109699E22

HF-X :

P0174319B7

P0174319B8

P0174319B9

Lumos:

P0109699K8

P0109699K9

P0109699K10

OrbiElite:

P81464G09

P81464G10

Raw Files are accesible via PRIDE:

<tba>

All benchmark files (igz) used in the paper can be found here:

https://drive.google.com/drive/folders/1GbN0cAqiyAEljcuooYKkqW4rcKlOlc5n

CHAPTER

TWO

INDICES AND TABLES

- genindex
- modindex
- search

PYTHON MODULE INDEX

p

INDEX

Symbols

A

C

E

F

G

H

I

M

N

P

R

S

T

W