Python in proteomics

Python is a versatile scripting language that is widely used in industry and academia. In bioinformatics, there are multiple packages supporting data analysis with Python that range from biological sequence analysis with Biopython to structural modeling and visualization with packages like PyMOL and PyRosetta, to numerical computation and advanced plotting with NumPy/SciPy. In the proteomics community, Python began to be widely used around 2012 when several mature Python packages were published including pymzML, Pyteomics and pyOpenMS. This has led to an ever-increasing interest in the Python programming language in the proteomics and mass spectrometry community. The number of publications referencing or using Python has risen eight fold since 2012 (compared with the same time period before 2012), with multiple open-source Python packages now supporting mass spectrometric data analysis and processing. Computing and data analysis in mass spectrometry is very diverse and in many cases must be tailored to a specific experiment. Often, multiple analysis steps have to be performed (identification, quantification, post-translational modification analysis, filtering, FDR analysis etc.) in an analysis pipeline, which requires high flexibility in the analysis. This is where Python truly shines, due to its flexibility, visualization capabilities and the ability to extend computation with a large number of powerful libraries. Python can be used to quickly prototype software, combine existing libraries into powerful analysis workflows while avoiding the trap of re- inventing the wheel for a new project.

Here, we will describe data analysis with Python using the pyOpenMS package. An extended documentation and tutorial can also be found online at https://pyopenms.readthedocs.io. To allow the reader to follow all steps in the tutorial, we will also describe the installation process of the software. Our installation is based on Anaconda, an open- source Python distribution that includes the Spyder integrated development environment (IDE) that allows development with pyOpenMS in a graphical environment.

PeerJ Preprints | https://doi.org/10.7287/peerj.preprints.27736v1 | CC BY 4.0 Open Access | rec: 16 May 2019, publ: 16 May 2019

Python in Proteomics

Hannes L. Röst[1]

[1]*Donnelly Centre, University of Toronto, Toronto, Canada. Contact: hannes.rost@utoronto.ca*

(Dated: May 15, 2019)

I. ABSTRACT

II. TEXT

Python is a versatile scripting language that is widely used in industry and academia. In bioinformatics, there are multiple packages supporting data analysis with Python that range from biological sequence analysis with Biopython [1] to structural modeling and visualization with packages like PyMOL and PyRosetta [2], to numerical computation and advanced plotting with NumPy/SciPy [3]. In the proteomics community, Python began to be widely used around 2012 when several mature Python packages were published including pymzML [4], Pyteomics [5] and pyOpenMS [6]. This has led to an ever-increasing interest in the Python programming language in the proteomics and mass spectrometry community. The number of publications referencing or using Python has risen eight fold since 2012 (compared with the same time period before 2012), with multiple open-source Python packages now supporting mass spectrometric data analysis and processing [4, 5, 7–14]. Computing and data analysis in mass spectrometry is very diverse and in many cases must be tailored to a specific experiment. Often, multiple analysis steps have to be performed (identification, quantification, post-translational modification analysis, filtering, FDR analysis *etc.*) in an analysis pipeline, which requires high flexibility in the analysis. This is where Python truly shines, due to its flexibility, visualization capabilities and the ability to extend computation with a large number of powerful libraries. Python can be used to quickly prototype software, combine existing libraries into powerful analysis workflows while avoiding the trap of re-inventing the wheel for a new project.

Here, we will describe data analysis with Python using the pyOpenMS package [6]. An extended documentation and tutorial can also be found online at `https://pyopenms.readthedocs.io`. To allow the reader to follow all steps in the tutorial, we will also describe the installation process of the software. Our installation is based on Anaconda, an open-source Python distribution that includes the Spyder integrated development environment (IDE) that allows development with pyOpenMS in a graphical environment.

III. INSTALLATION

In order to install pyOpenMS, we recommend to use Anaconda and Spyder which provide a fully integrated environment for developing, running and debugging Python code. You can download the latest Python version (currently 3.7) bundled with Anaconda from `https://www.anaconda.com/distribution/`. After installation, you will have multiple new applications available on your computer. To set up pyOpenMS, use the newly installed Anaconda environment to install pyOpenMS (on Windows: go to the Start Menu and click on the "Anaconda Powershell Prompt" application) and type:

```
pip install pyopenms
```

which will result in a message "Successfully installed pyopenms" including a version number. You should now be set up to use pyOpenMS and you can start the Spyder application (on Windows: go to the Start Menu and click on the "Spyder" application) which will open up and provide you with multiple windows (see Fig. 1).

In addition, we recommend that you install the OpenMS tool suite from `https://www.openms.de/download/openms-binaries/` which will install the visualization tool TOPPView as well as over 180 TOPP tools that are executables distributed with OpenMS (see chapter YYY for more information on OpenMS and TOPPView).

IV. GETTING STARTED

After installation, you can use the full extent of the OpenMS library. There are multiple ways to get information about the available functions and methods. We can inspect individual pyOpenMS objects through the `help` function:

```
from pyopenms import *
help(MSExperiment)
```

which will display an extensive help text and list all available functions, indicating that `MSExperiment` exposes methods such as `getNrSpectra()` and `getSpectrum(id)` where the argument `id` indicates the spectrum identifer. It also points to the base class `ExperimentalSettings` which can be investigated in the same manner for additional information.

pyOpenMS supports a variety of different files through the implementations in OpenMS. In order to demonstrate the capabilities of pyOpenMS to read different mass spectrometric data files, we will download two files that have been prepared for this chapter and are available from Zenodo[1]:

```
from urllib.request import urlretrieve
from pyopenms import *
url = "https://zenodo.org/record/2653155/files/"
urlretrieve (url + "example.mzML", "example.mzML")
urlretrieve (url + "search.fasta", "search.fasta")
exp = MSExperiment()
MzMLFile().load("example.mzML", exp)
exp.getNrSpectra()
exp.getNrChromatograms()
```

which will load the content of the "example.mzML" file into the `exp` variable of type `MSExperiment`. The file contains 3 spectra and 5 chromatograms, which we can see from the output of the code above.

V. PLOTTING

pyOpenMS has basic functionality to plot spectra and chromatograms, which we can now try out on our file:

```
from pyopenms import *
exp = MSExperiment()
MzMLFile().load("example.mzML", exp)
Plot.plotSpectrum(exp.getSpectrum(0))
Plot.plotChromatogram(exp.getChromatogram(0))
```

This will generally produce an interactive plot of the spectrum and the chromatogram, which allows zooming, panning and detailed manual analysis of the data. When using Spyder, by default an inline plot will be generated that is static and cannot be manipulated.

[1] `https://zenodo.org/record/2653155`

VI. CHEMISTRY

A. Elements

OpenMS has representations for various chemical concepts including molecular formulas, isotopes, amino acid sequences and modifications. First, we look at how elements are stored in OpenMS:

```
from pyopenms import *

edb = ElementDB()
sulfur = edb.getElement("S")
print(sulfur.getName())
isotopes = sulfur.getIsotopeDistribution()
for iso in isotopes.getContainer():
    print (iso.getMZ(), ":", iso.getIntensity())
```

As we can see, OpenMS knows common elements like Sulfur as well as their isotopic distribution. These values are stored in `Elements.xml` in the OpenMS share folder and can, in principle, be modified. The above code outputs the isotopes of sulfur and their abundance.

B. Molecular Formula

Elements can be combined to molecular formulas (`EmpiricalFormula`) which can be used to describe small molecules or peptides. The class supports a large number of operations like addition and subtraction. A simple example is given in the next few lines of code.

```
from pyopenms import *

methanol = EmpiricalFormula("CH3OH")
water = EmpiricalFormula("H2O")
ethanol = EmpiricalFormula("CH2" + methanol.toString())
wm = water + methanol
```

```
print(wm.toString())
print(wm.getElementalComposition())
```

Note how in lines 5 and 6 we were able to make new molecules by adding existing molecules (either by adding two `EmpiricalFormula` objects or by adding simple strings).

C. Isotopic Distributions

OpenMS can also generate theoretical isotopic distributions from analytes represented as `EmpiricalFormula`. Currently there are two algorithms implemented, CoarseIsotopePatternGenerator which produces unit mass isotope patterns and FineIsotopePatternGenerator which is based on the IsoSpec algorithm[15]:

```
from pyopenms import *

wm = EmpiricalFormula("CH3OH") + EmpiricalFormula("H2O")

print("Coarse Isotope Distribution:")
gen = CoarseIsotopePatternGenerator(5)
isotopes = wm.getIsotopeDistribution(gen)
for iso in isotopes.getContainer():
    print (iso.getMZ(), ":", iso.getIntensity())

print("Fine Isotope Distribution:")
gen = FineIsotopePatternGenerator(1e-5)
isotopes = wm.getIsotopeDistribution(gen)
for iso in isotopes.getContainer():
    print (iso.getMZ(), ":", iso.getIntensity())
```

Note how the result calculated with the `FineIsotopePatternGenerator` contains the hyperfine isotope structure with heavy isotopes of Carbon, Hydrogen and Oxygen clearly distinguished while the coarse (unit resolution) isotopic distribution contains summed probabilities for each isotopic peak without the hyperfine resolution. Also note how the differences between the hyperfine peaks can reach more than 115 ppm (52.041 vs 52.047). Note that

the FineIsotopePatternGenerator will generate peaks until the total probability not covered by the current result reaches 1e-5.

D. Amino Acids

An amino acid residue is represented in OpenMS by the class `Residue`. It provides a container for the amino acids as well as some functionality. The class is able to provide information such as the isotope distribution of the residue, the average and monoisotopic weight. The residues can be identified by their full name, their three letter abbreviation or the single letter abbreviation. The residue can also be modified, which is implemented in the `Modification` class.

An amino acid residue modification is represented in OpenMS by the class `ResidueModification`. The known modifications are stored in the `ModificationsDB` object, which is capable of retrieving specific modifications. It contains UniMod as well as PSI modifications.

VII. PEPTIDES AND PROTEINS

A. Amino Acid Sequences

The `AASequence` class handles amino acid sequences in OpenMS. A string of amino acid residues can be turned into a instance of `AASequence` to provide some commonly used operations and data. The implementation supports mathematical operations like addition or subtraction. Also, average and mono isotopic weight and isotope distributions are accessible.

Weights, formulas and isotope distribution can be calculated depending on the charge state (additional proton count in case of positive ions) and ion type. Therefore, the class allows for a flexible handling of amino acid strings.

A very simple example of handling amino acid sequence with AASequence is given in the next few lines, which also calculates the weight of the `(M)` and `(M+2H)2+` ions.

```
from pyopenms import *
seq = AASequence.fromString("DFPIANGER")
concat = seq + seq
```

```
# weight of M
seq.getMonoWeight()
# weight of M+2H
seq.getMonoWeight(Residue.ResidueType.Full, 2)
mz = seq.getMonoWeight(Residue.ResidueType.Full, 2) / 2.0
concat.getMonoWeight()

print("Monoisotopic m/z of (M+2H)2+ is", mz)
```

B. Molecular Formula

Note howe we can easily calculate the charged weight of a `(M+2H)2+` ion on line 11 and compute m/z on line 12 – simply dividing by the charge. We can now combine our knowledge of `AASequence` with what we learned above about `EmpiricalFormula` to get accurate mass and isotope distributions from the amino acid sequence:

```
from pyopenms import *
seq = AASequence.fromString("DFPIANGER")
seq_formula = seq.getFormula()
print("Peptide", seq, "has molecular formula", seq_formula)
print("="*35)

gen = CoarseIsotopePatternGenerator(6)
isotopes = seq_formula.getIsotopeDistribution(gen)
for iso in isotopes.getContainer():
  print ("Isotope", iso.getMZ(), ":", iso.getIntensity())

suffix = seq.getSuffix(3) # y3 ion "GER"
print("="*35)
print("y3 ion :", suffix)
y3_formula = suffix.getFormula(Residue.ResidueType.YIon, 2) # y3++ ion
suffix.getMonoWeight(Residue.ResidueType.YIon, 2) / 2.0 # CORRECT
```

```
suffix.getMonoWeight(Residue.ResidueType.XIon, 2) / 2.0 # CORRECT
suffix.getMonoWeight(Residue.ResidueType.BIon, 2) / 2.0 # INCORRECT

print("y3 mz :", suffix.getMonoWeight(Residue.ResidueType.YIon, 2) / 2.0 )
print(y3_formula)
print(seq_formula)
```

Note on lines 13 to 15 we need to remember that we are dealing with an ion of the x/y/z series since we used a suffix of the original peptide and using any other ion type will produce a different mass-to-charge ratio (and while "GER" would also be a valid "x3" ion, note that it *cannot* be a valid ion from the a/b/c series and therefore the mass on line 15 cannot refer to the same input peptide "DFPIANGER" since its "b3" ion would be "DFP" and not "GER").

C. Modified Sequences

The `AASequence` class can also handle modifications, modifications are specified using a unique string identifier present in the `ModificationsDB` in round brackets after the modified amino acid or by providing the mass of the residue in square brackets. For example `AASequence.fromString(".DFPIAM(Oxidation)GER.")` creates an instance of the peptide "DFPIAMGER" with an oxidized methionine. There are multiple ways to specify modifications, and `AASequence.fromString("DFPIAM(UniMod:35)GER")`, `AASequence.fromString("DFPIAM[+16]GER")` and `AASequence.fromString("DFPIAM[147]GER")` are all equivalent).

```
from pyopenms import *
seq = AASequence.fromString("PEPTIDESEKUEM(Oxidation)CER")
print(seq.toUnmodifiedString())
print(seq.toString())
print(seq.toUniModString())
print(seq.toBracketString())
print(seq.toBracketString(False))
```

```
print(AASequence.fromString("DFPIAM(UniMod:35)GER"))
print(AASequence.fromString("DFPIAM[+16]GER"))
print(AASequence.fromString("DFPIAM[+15.99]GER"))
print(AASequence.fromString("DFPIAM[147]GER"))
print(AASequence.fromString("DFPIAM[147.035405]GER"))
```

Note there is a subtle difference between `AASequence.fromString(".DFPIAM[+16]GER.")` and `AASequence.fromString(".DFPIAM[+15.9949]GER.")` - while the former will try to find the first modification matching to a mass difference of 16 +/- 0.5, the latter will try to find the closest matching modification to the exact mass. The exact mass approach usually gives the intended results while the first approach may or may not.

D. Proteins

Protein sequences can be accessed through the `FASTAEntry` object and can be read and stored on disk using a `FASTAFile`:

```
from pyopenms import *
bsa = FASTAEntry()
bsa.sequence = "MKWVTFISLLLLFSSAYSRGVFRRDTHKSEIAHRFKDLGE"
bsa.description = "BSA Bovine Albumin (partial sequence)"
bsa.identifier = "BSA"
alb = FASTAEntry()
alb.sequence = "MKWVTFISLLFLFSSAYSRGVFRRDAHKSEVAHRFKDLGE"
alb.description = "ALB Human Albumin (partial sequence)"
alb.identifier = "ALB"

entries = [bsa, alb]

f = FASTAFile()
f.store("example.fasta", entries)
```

E. TheoreticalSpectrumGenerator

This class implements a simple generator which generates tandem MS spectra from a given peptide charge combination. There are various options which influence the occurring ions and their intensities.

```
from pyopenms import *

tsg = TheoreticalSpectrumGenerator()
spec1 = MSSpectrum()
spec2 = MSSpectrum()
peptide = AASequence.fromString("DFPIANGER")
# standard behavior is adding b- and y-ions of charge 1
p = Param()
p.setValue("add_b_ions", "false")
tsg.setParameters(p)
tsg.getSpectrum(spec1, peptide, 1, 1)
p.setValue("add_b_ions", "true")
p.setValue("add_a_ions", "true")
p.setValue("add_losses", "true")
p.setValue("add_metainfo", "true")
tsg.setParameters(p)
tsg.getSpectrum(spec2, peptide, 1, 2)
print("Spectrum 1 has", spec1.size(), "peaks.")
print("Spectrum 2 has", spec2.size(), "peaks.")

# Iterate over annotated ions and their masses
for ion, peak in zip(spec2.getStringDataArrays()[0], spec2):
    print(ion, peak.getMZ())
```

The example shows how to put peaks of a certain type, y-ions in this case, into a spectrum. Spectrum 2 is filled with a complete spectrum of all peaks (a-, b-, y-ions and losses). The

`TheoreticalSpectrumGenerator` has many parameters which have a detailed description located in the class documentation. For the first spectrum, no b ions are added. Note how the `add_metainfo` parameter in the second example populates the `StringDataArray` of the output spectrum, allowing us to iterate over annotated ions and their masses.

VIII. DIGESTION

A. Proteolytic Digestion with Trypsin

OpenMS has classes for proteolytic digestion which can be used as follows:

```
from pyopenms import *
from urllib.request import urlretrieve
urlretrieve ("http://www.uniprot.org/uniprot/P02769.fasta", "bsa.fasta")

dig = ProteaseDigestion()
dig.getEnzymeName() # Trypsin
bsa = "".join([l.strip() for l in open("bsa.fasta").readlines()[1:]])
bsa = AASequence.fromString(bsa)
result = []
dig.digest(bsa, result)
print(result[4].toString())
len(result) # 82 peptides
```

B. Proteolytic Digestion with Lys-C

We can of course also use different enzymes, these are defined `Enzyme.xml` file and can be accessed using the `EnzymesDB`

```
names = []
ProteaseDB().getAllNames(names)
len(names) # at least 25 by default
e = ProteaseDB().getEnzyme('Lys-C')
```

```
e.getRegExDescription()
e.getRegEx()
```

Now that we have learned about the other enzymes available, we can use it to cut out protein of interest:

```
from pyopenms import *
from urllib.request import urlretrieve
urlretrieve ("http://www.uniprot.org/uniprot/P02769.fasta", "bsa.fasta")

dig = ProteaseDigestion()
dig.setEnzyme('Lys-C')
bsa = "".join([l.strip() for l in open("bsa.fasta").readlines()[1:]])
bsa = AASequence.fromString(bsa)
result = []
dig.digest(bsa, result)
print(result[4].toString())
len(result) # 57 peptides
```

We now get different digested peptides (57 vs 82) and the fourth peptide is now `GLVLIAFSQYLQQCPFDEHVK` instead of `DTHK` as with Trypsin (see above).

IX. SIMPLE DATA MANIPULATION

Here we will look at a few simple data manipulation techniques on spectral data, such as filtering.

A. Filtering Spectra

We will filter the "example.mzML" file by only retaining spectra that match a certain identifier:

```
from pyopenms import *
inp = MSExperiment()
```

```
MzMLFile().load("example.mzML", inp)

e = MSExperiment()
for s in inp:
  if s.getNativeID().startswith("scan="):
    e.addSpectrum(s)

MzMLFile().store("test_filtered.mzML", e)
```

B. Filtering by MS level

Similarly, we can filter the `example.mzML` file by MS level, retaining only spectra that are not MS1 spectra (e.g. MS2, MS3 or MSn spectra):

```
from pyopenms import *
inp = MSExperiment()
MzMLFile().load("example.mzML", inp)

e = MSExperiment()
for s in inp:
  if s.getMSLevel() > 1:
    e.addSpectrum(s)

MzMLFile().store("test_filtered.mzML", e)
```

Note that we can easily replace line 7 with more complicated criteria, such as filtering by MS level and scan identifier at the same time:

```
if s.getMSLevel() > 1 and s.getNativeID().startswith("scan="):
```

C. Filtering by scan number

Or we could use an external list of scan numbers to filter by scan numbers, thus only retaining MS scans in which we are interested in:

```
from pyopenms import *
inp = MSExperiment()
MzMLFile().load("example.mzML", inp)
scan_nrs = [0, 2, 5, 7]

e = MSExperiment()
for k, s in enumerate(inp):
  if k in scan_nrs and s.getMSLevel() == 1:
    e.addSpectrum(s)

MzMLFile().store("test_filtered.mzML", e)
```

D. Filtering Spectra and Peaks

We can now move on to more advanced filtering, suppose we are interested in only a part of all fragment ion spectra, such as a specific m/z window. We can easily filter our data accordingly:

```
from pyopenms import *
inp = MSExperiment()
MzMLFile().load("example.mzML", inp)

mz_start = 6.0
mz_end = 12.0
e = MSExperiment()
for s in inp:
  if s.getMSLevel() > 1:
    filtered_mz = []
    filtered_int = []
    for mz, i in zip(*s.get_peaks()):
      if mz > mz_start and mz < mz_end:
        filtered_mz.append(mz)
        filtered_int.append(i)
```

```
    s.set_peaks((filtered_mz, filtered_int))
    e.addSpectrum(s)

MzMLFile().store("test_filtered.mzML", e)
```

Note that in a real-world application, we would set the `mz_start` and `mz_end` parameter to an actual area of interest, for example the area between 125 and 132 which contains quantitative ions for a TMT experiment.

Similarly we could change line 13 to only report peaks above a certain intensity or to only report the top N peaks in a spectrum.

E. Memory management

On order to save memory, we can avoid loading the whole file into memory and use the OnDiscMSExperiment for reading data.

```
from pyopenms import *
ondisc_exp = OnDiscMSExperiment()
ondisc_exp.openFile("example.mzML")

e = MSExperiment()
for k in range(ondisc_exp.getNrSpectra()):
  s = ondisc_exp.getSpectrum(k)
  if s.getNativeID().startswith("scan="):
    e.addSpectrum(s)

MzMLFile().store("test_filtered.mzML", e)
```

Note that using the approach the output data `e` is still completely in memory and may end up using a substantial amount of memory. We can avoid that by using the following code:

```
from pyopenms import *
ondisc_exp = OnDiscMSExperiment()
```

```
ondisc_exp.openFile("example.mzML")

consumer = PlainMSDataWritingConsumer("test_filtered.mzML")

e = MSExperiment()
for k in range(ondisc_exp.getNrSpectra()):
  s = ondisc_exp.getSpectrum(k)
  if s.getNativeID().startswith("scan="):
    consumer.consumeSpectrum(s)

del consumer
```

Make sure you do not forget `del consumer` since otherwise the final part of the mzML may not get written to disk (and the consumer is still waiting for new data).

X. EXAMPLE: PEPTIDE SEARCH

In MS-based proteomics, fragment ion spectra (MS2 spectra) are often interpreted by comparing them against a theoretical set of spectra generated from a FASTA database. OpenMS contains a (simple) implementation of such a "search engine" that compares experimental spectra against theoretical spectra generated from a chemical or enzymatic digest of a proteome.

In most proteomics applications, a dedicated search engine (such as Comet, Crux, Mascot, MSGFPlus, MSFragger, Myrimatch, OMSSA, SpectraST or XTandem; all of which are supported by pyOpenMS) will be used to search data. Here, we will use the internal `SimpleSearchEngineAlgorithm` from OpenMS used for teaching purposes. This makes it very easy to search an (experimental) mzML file against a fasta database of protein sequences:

```
from pyopenms import *
SimpleSearchEngineAlgorithm().search("example.mzML",
  "search.fasta", protein_ids, peptide_ids)
```

This will print search engine output including the number of peptides and proteins in the database and how many spectra were matched to peptides and proteins. We can now investigate the individual peptide spectrum matches (PSM) using Python:

```
for peptide_id in peptide_ids:
  # Peptide identification values
  print (35*"=")
  print ("Peptide ID m/z:", peptide_id.getMZ())
  print ("Peptide ID rt:", peptide_id.getRT())
  print ("Peptide scan index:", peptide_id.getMetaValue("scan_index"))
  print ("Peptide scan name:", peptide_id.getMetaValue("scan_index"))
  print ("Peptide ID score type:", peptide_id.getScoreType())
  # PeptideHits
  for hit in peptide_id.getHits():
    print(" - Peptide hit rank:", hit.getRank())
    print(" - Peptide hit charge:", hit.getCharge())
    print(" - Peptide hit sequence:", hit.getSequence())
    z = hit.getCharge()
    mz = hit.getSequence().getMonoWeight(Residue.ResidueType.Full, z) / z
    print(" - Peptide hit monoisotopic m/z:", mz)
    print(" - Peptide ppm error:", abs(mz - peptide_id.getMZ())/mz *10**6 )
    print(" - Peptide hit score:", hit.getScore())
```

We notice that the second peptide spectrum match (PSM) was found for the third spectrum in the file for a precursor at 775.38 m/z for the sequence `RPGADSDIGGFGGLFDLAQAGFR`.

```
tsg = TheoreticalSpectrumGenerator()
thspec = MSSpectrum()
p = Param()
p.setValue("add_metainfo", "true")
tsg.setParameters(p)
peptide = AASequence.fromString("RPGADSDIGGFGGLFDLAQAGFR")
tsg.getSpectrum(thspec, peptide, 1, 1)
```

```
# Iterate over annotated ions and their masses
for ion, peak in zip(thspec.getStringDataArrays()[0], thspec):
    print(ion, peak.getMZ())


e = MSExperiment()
MzMLFile().load("searchfile.mzML", e)
print ("Spectrum native id", e[2].getNativeID() )
mz,i = e[2].get_peaks()
peaks = [(mz,i) for mz,i in zip(mz,i) if i > 1500 and mz > 300]
for peak in peaks:
  print (peak[0], "mz", peak[1], "int")
```

Comparing the theoretical spectrum and the experimental spectrum for `RPGADSDIGGFGGLFDLAQAGFR` we can easily see that the most abundant ions in the spectrum are y8 (877.452 m/z), b10 (926.432), y9 (1024.522 m/z) and b13 (1187.544 m/z).

We can now use the OpenMS tool TOPPView for visualization of the mzML file (see chapter YYY for more information on OpenMS and TOPPView). When loading the `searchfile.mzML` into the OpenMS visualization software TOPPView, we can convince ourselves that the observed spectrum indeed was generated by the peptide `RPGADSDIGGFGGLFDLAQAGFR` by loading the corresponding theoretical spectrum into the viewer using "Tools"->"Generate theoretical spectrum" (see Fig. 2).

XI. PYOPENMS IN R

The R programming language is a powerful open-source statistical programming language that is often used in Bioinformatics. While chapter XXX describes the available tools in R in greater detail, here we will briefly discuss how one can use an existing Python library, such as pyOpenMS, directly in R. Since there are no native wrappers for the OpenMS library in R, we will use the "reticulate" package in order to get access to the full functionality of pyOpenMS in the R programming language.

A. Install the "reticulate" R package

In order to use all pyopenms functionalities in R, we suggest to use the "reticulate" R package.

A thorough documentation is available at: https://rstudio.github.io/reticulate/

```
install.packages("reticulate")
```

Installation of pyopenms is a requirement as well and it is necessary to make sure that R is using the same python environment.

In case R is having trouble to find the correct Python environment, you can set it by hand as in this example (using miniconda, you will have to adjust the file path to your system to make this work). You will need to do this before loading the "reticulate" library:

```
Sys.setenv(RETICULATE_PYTHON = "/usr/local/miniconda3/envs/py37/bin/python")
```

Or after loading the "reticulate" library:

```
library("reticulate")
use_python("/usr/local/miniconda3/envs/py37/bin/python")
```

B. Import pyopenms in R

After loading the "reticulate" library you should be able to import pyopenms into R

```
library(reticulate)
ropenms=import("pyopenms", convert = FALSE)
```

This should now give you access to all of pyopenms in R. Importantly, the convert option has to be set to FALSE, since type conversions such as 64bit integers will cause problems.

You should now be able to interact with the OpenMS library and, for example, read and write mzML files:

```
library(reticulate)
ropenms=import("pyopenms", convert = FALSE)
exp = ropenms$MSExperiment()
ropenms$MzMLFile()$store("testfile.mzML", exp)
```

which will create an empty mzML file called *testfile.mzML*.

---

[1] Cock, P. J., et al. (2009) Biopython: freely available python tools for computational molecular biology and bioinformatics. *Bioinformatics*, **25**, 1422–1423.

[2] Chaudhury, S., Lyskov, S., and Gray, J. J. (2010) PyRosetta: a script-based interface for implementing molecular modeling algorithms using rosetta. *Bioinformatics*, **26**, 689–691.

[3] Jones, E., Oliphant, T., Peterson, P., and Others (2001) *SciPy: Open source scientific tools for Python*.

[4] Bald, T., Barth, J., Niehues, A., Specht, M., Hippler, M., and Fufezan, C. (2012) pymzml—python module for high-throughput bioinformatics on mass spectrometry data. *Bioinformatics*, **28**, 1052–1053.

[5] Goloborodko, A. A., Levitsky, L. I., Ivanov, M. V., and Gorshkov, M. V. (2013) Pyteomics—a python framework for exploratory data analysis and rapid software prototyping in proteomics. *Journal of The American Society for Mass Spectrometry*, **24**, 301–304.

[6] Röst, H. L., Schmitt, U., Aebersold, R., and Malmström, L. (2014) pyOpenMS: A python-based interface to the OpenMS mass-spectrometry algorithm library. *Proteomics*, **14**, 74–77.

[7] Kiefer, P., Schmitt, U., and Vorholt, J. A. (2013) eMZed: an open source framework in python for rapid and interactive development of LC/MS data analysis workflows. *Bioinformatics (Oxford, England)*, **29**, 963–964.

[8] Alexander, W. M., Ficarro, S. B., Adelmant, G., and Marto, J. A. (2017) multiplierz v2. 0: A python-based ecosystem for shared access and analysis of native mass spectrometry data. *Proteomics*, **17**, 1700091.

[9] Levitsky, L. I., Klein, J. A., Ivanov, M. V., and Gorshkov, M. V. (2018) Pyteomics 4.0: five years of development of a python proteomics framework. *Journal of proteome research*, **18**, 709–714.

[10] Kösters, M., Leufken, J., Schulze, S., Sugimoto, K., Klein, J., Zahedi, R., Hippler, M., Leidel, S., and Fufezan, C. (2018) pymzml v2. 0: introducing a highly compressed and seekable gzip format. *Bioinformatics*, **34**, 2513–2514.

[11] Ressa, A., Fitzpatrick, M., Van den Toorn, H., Heck, A. J., and Altelaar, M. (2018) Padua: A python library for high-throughput (phospho) proteomics data analysis. *Journal of proteome*

research, **18**, 576–584.

[12] Klein, J. and Zaia, J. (2019) psims-a declarative writer for mzml and mzidentml for python. *Molecular & Cellular Proteomics*, **18**, 571–575.

[13] May, D. H., Tamura, K., and Noble, W. S. (2019) Detecting modifications in proteomics experiments with param-medic. *Journal of proteome research*.

[14] Yunker, L. P., Donnecke, S., Ting, M., Yeung, D., and McIndoe, J. S. (2019) Pythoms: A python framework to simplify and assist in the processing and interpretation of mass spectrometric data. *Journal of chemical information and modeling*.

[15] Łacki, M. K., Startek, M., Valkenborg, D., and Gambin, A. (2017) Isospec: hyperfast fine structure calculator. *Analytical chemistry*, **89**, 3272–3277.

XII. ACKNOWLEDGEMENTS

This Project was funded by the Government of Canada through Genome Canada and the Ontario Genomics Institute (OGI-164). This work was supported by the Canadian Institutes for Health Research, the Natural Sciences and Engineering Research Council of Canada, and the Canada Research Coordinating Committee. H.L.R. is supported by the Canadian Foundation for Innovation and the John R. Evans Leaders Fund and is the Canada Research Chair in Mass Spectrometry-based Personalized Medicine.

Figures

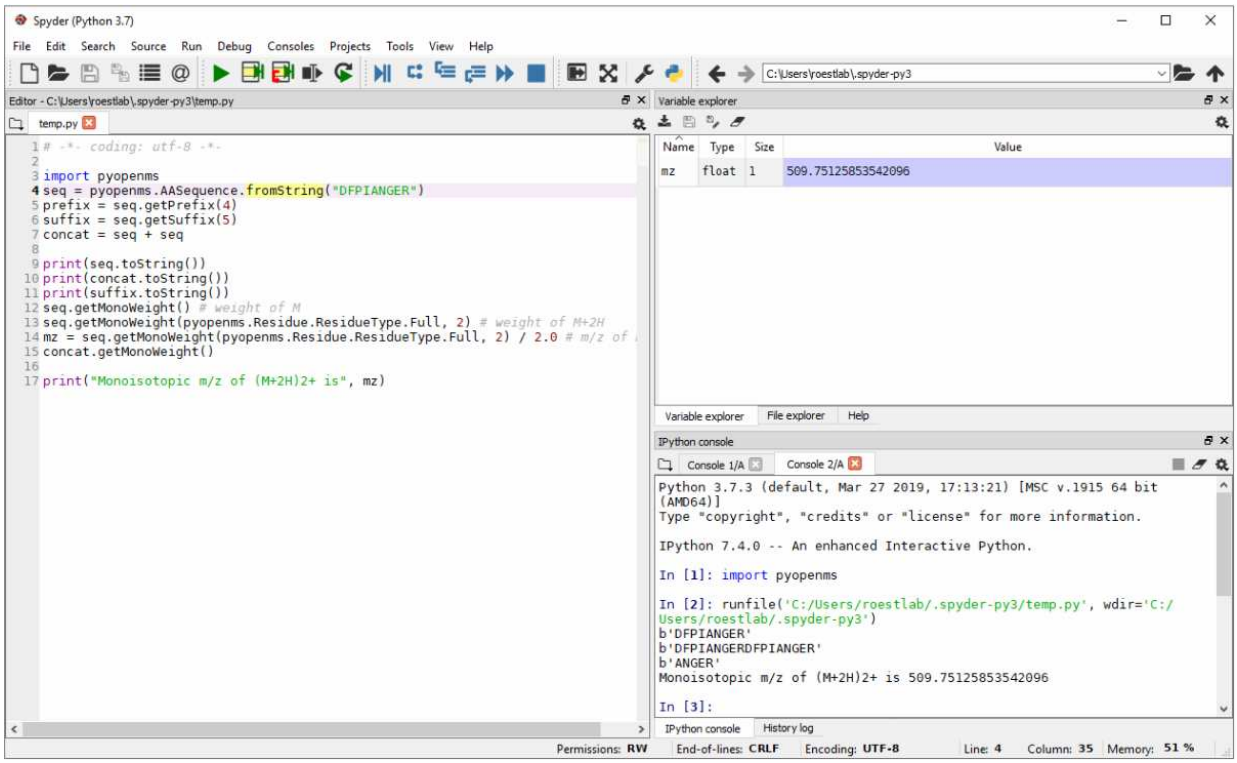

FIG. 1: **Spyder graphical interface for Python.** The Spyder software is an open-source integrated development environment (IDE) for Python which allows users to develop, run and debug Python scripts. Here, a screenshot of the software is shown after executing the `import pyopenms` command in the console (lower right, see `In [1]`). In the left window, a script that is currently active is shown and its result is shown on the command window (see `In [2]`). On the top, the variable explorer is active which shows the currently available variables, here the `mz` variable is displayed that has the value 509.75 and is computed on line 14 of the script.

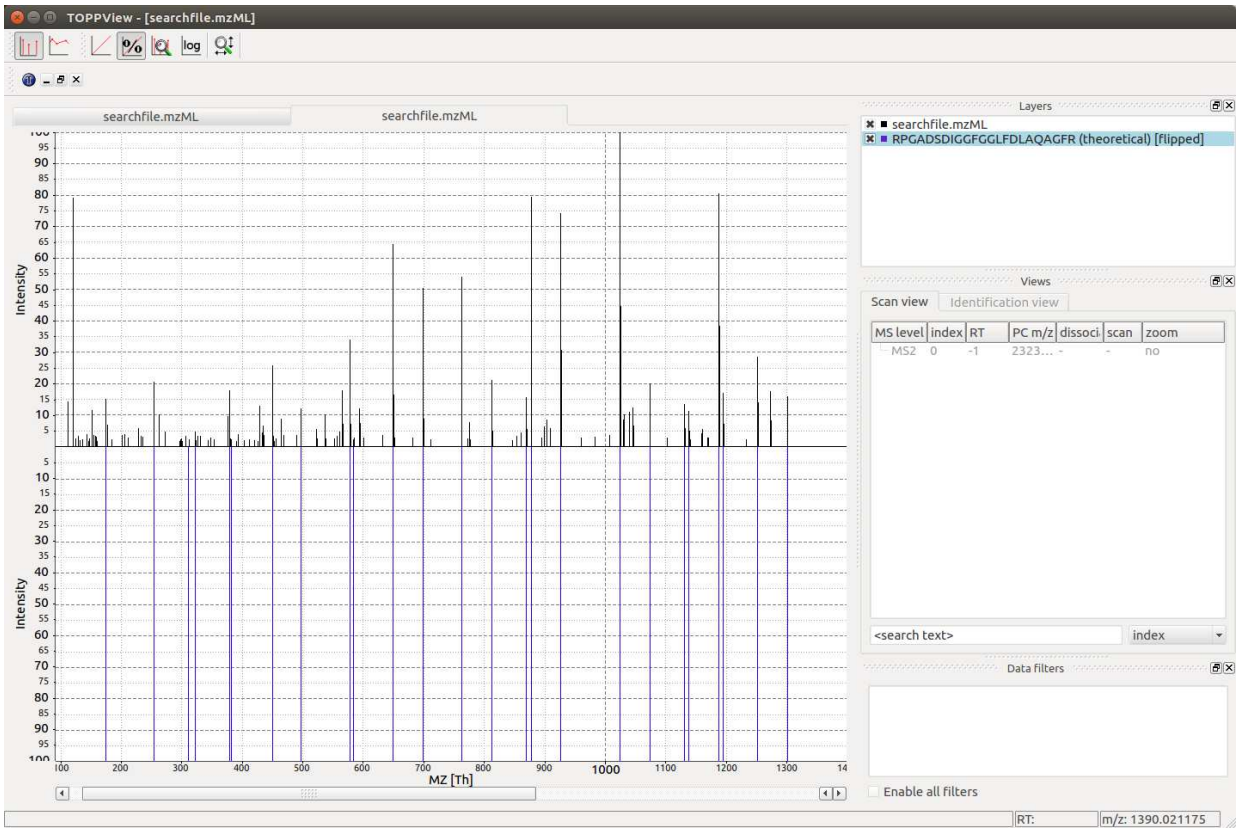

FIG. 2: **Peptide spectrum match, visualized with TOPPView.** A theoretical spectrum and an experimental spectrum are visualized together using the OpenMS software TOPPView.