chemfp Documentation

Release 3.1

Andrew Dalke

Sep 18, 2017

Contents

chemfp is a set of tools for working with cheminformatics fingerprints.

This is the documentation for the commerical version of chemfp. The documentation for chemfp 1.3, the no-cost version of chemfp, is available from http://chemfp.readthedocs.io/en/chemfp-1.3/.

Most people will use the command-line programs to generate and search fingerprint files. *ob2fps*, *oe2fps*, and *rdkit2fps* use respectively the Open Babel, OpenEye, and RDKit chemistry toolkits to convert structure files into fingerprint files. *sdf2fps* extracts fingerprints encoded in SD tags to make the fingerprint file. *simsearch* finds targets in a fingerprint file which are sufficiently similar to the queries. *fpcat* converts between FPS and FPB formats and merges multiple fingerprint files into one.

The programs are built using the *chemfp Python library API*. The search capabilities are part of the public API, as well as a cross-toolkit API for reading and writing molecules from structure files or strings, and for computing molecular fingerprints.

Remember: chemfp cannot generate fingerprints from a structure file without a third-party chemistry toolkit.

Chemfp 3.1 was released on 18 September 2017. It supports Python 2.7 and 3.5+ and can be used with any recent version of OEChem/OEGraphSim, Open Babel, or RDKit.

CHAPTER 1

List of chapters

Installing

Chemfp requires that Python and a C compiler be installed in your machines. Since chemfp doesn't run on Microsoft Windows (for tedious technical reasons), then your machine likely already has both Python and a C compiler installed. In case you don't have Python, or you want to install a newer version, you can download a copy of Python from http://www.python.org/download/ . If you don't have a C compiler, .. well, do I really need to give you a pointer for that?

You may use chemfp 3.1 with either Python 2.7, or Python 3.5 or newer. If you want to run on Python 2.6 then you'll need to use chemfp 2.1.

The core chemfp functionality does not depend on a third-party library but you will need a chemistry toolkit in order to generate new fingerprints from structure files. chemfp supports the free Open Babel and RDKit toolkits and the proprietary OEChem toolkit. Make sure you install the Python libraries for the toolkit(s) you select.

The easiest way to install chemfp is with the pip installer. This comes with Python 2.7.9 or later, and with Python 3.4 and later so it may already be installed. To install the `tar.gz` file with pip:

```
pip install chemfp-3.1.tar.gz
```

Otherwise you can use Python's standard "setup.py". Read http://docs.python.org/install/index.html for details of how to use it. The short version is to do the following:

```
tar xf chemfp-3.1.tar.gz
cd chemfp-3.1
python setup.py build
python setup.py install
```

The last step may need a `sudo` if you otherwise cannot write to your Python site-package. Another option is to use a virtual environment.

Configuration options

The setup.py file has several compile-time options which can be set either from the `python setup.py build` command-line or through environment variables. The environment variable solution is the easiest way to change the settings under pip.

`--with-openmp, --without-openmp`

Chemfp uses OpenMP to parallelize multi-query searches. The default is *`--with-openmp`*. If you have a very old version of gcc, or an older version of clang, or are on a Mac where the clang version doesn't support OpenMP, then you will need to use *`--without-openmp`* to tell setup.py to compile without OpenMP:

```
python setup.py build --without-openmp
```

You can also set the environment variable CHEMFP_OPENMP to "1" to compile with OpenMP support, or to "0" to compile without OpenMP support:

```
CHEMFP_OPENMP=0 pip install chemfp-3.1.tar.gz
```

Note: you can use the environment variable `CC` to change the C compiler. For example, the clang compiler on Mac doesn't support OpenMP so I installed gcc-7 and compile using:

```
CC=gcc-7 pip install chemfp-3.1.tar.gz
```

`--with-ssse3, --without-ssse3`

Chemfp by default compiles with SSSE3 support, which was first available in 2006 so almost certainly available on your Intel-like processor. In case I'm wrong (are you compiling for ARM? If so, send my any compiler patches), you can disable SSSE3 support using the *`--without-ssse3`*, or set the environment variable `CHEMFP_SSSE3` to "0".

Compiling with SSSE3 support has a very odd failure case. If you compile with the SSSE3 flag enabled, then take the binary to a machine without SSSE3 support, then it will crash because all of the code will be compiled to expect the SSSE3 instruction set even though only one file, popcount_SSSE3.c, should be compiledthat way.

The solution is to compile popcount_SSSE3.c with the SSSE3 flag enabled and all of the other files without that flag. Unfortunately, Python's setup.py doesn't make that easy to do. If this is a problem for you, take a look at `filter_gcc` in the chemfp distribution. It's used like this:

```
CC=$PWD/filter_gcc python setup.py build
```

It's a bit of a hack so contact me if you have problems.

`--with-avx2, --without-avx2`

Chemfp 3.0 added support for the AVX2 instruction set. This can be 15% faster than the POPCNT instruction for large (ie, 2048 bit or greater) fingerprints. By default it is disabled. Use *`--with-avx2`* or set the environment variable `CHEMFP_AVX2` to "1" to enable it.

While 15% faster sounds great, I have only tested the AVX2 support in one machine environment. I expect that it will have similar portability problems as the SSSE3 code had, that is, if the code is compiled with the AVX2 compilation flag then it's free to assume that some other instruction sets, like SSE4.2, are also available. Because of the way Python's setup.py works, all of the code will be compiled to use these more advanced instructions. If chemfp is then run on a machine without those instructions, it will cause the program to crash with an illegal instruction.

Chemfp does check that the chip implements AVX2 before calling the functions which are explicitly written with AVX2. The problem is that other parts of the code may be affected, at the compiler's disgression. I have no way of knowing.

A solution is to modify the `filter_gcc` option I mentioned earlier. Let me know if this is something you want me to work on with you.

Working with the command-line tools

The sections in this chapter describe examples of using the command-line tools to generate fingerprint files and to do similarity searches of those files.

Generate fingerprint files from PubChem SD tags

In this section you'll learn how to create a fingerprint file from an SD file which contains pre-computed CACTVS fingerprints. You do not need a chemistry toolkit for this section.

PubChem is a great resource of publically available chemistry information. The data is available for ftp download. We'll use some of their SD formatted files. Each record has a PubChem/CACTVS fingerprint field, which we'll extract to generate an FPS file.

Start by downloading the files Compound_027575001_027600000.sdf.gz and Compound_014550001_014575000.sdf.gz. At the time of writing (April 2017) they contain 384 and 5167 records, respectively. (I chose smaller than average files so they would be easier to open and review.)

Next, convert the files into fingerprint files. On the command line do the following two commands:

```
sdf2fps --pubchem Compound_027575001_027600000.sdf.gz -o pubchem_queries.fps
sdf2fps --pubchem Compound_014550001_014575000.sdf.gz -o pubchem_targets.fps
```

Congratulations, that was it!

If you're curious about what an FPS file looks like, here are the first 10 lines of pubchem_queries.fps, with some of the lengthy fingerprint lines replaced with an ellipsis:

```
#FPS1
#num_bits=881
#type=CACTVS-E_SCREEN/1.0 extended=2
#software=CACTVS/unknown
#source=Compound_027575001_027600000.sdf.gz
#date=2017-09-16T21:25:08
075e1c00020800000000 ... 1fd7e9191304710000040200200100000002010090000000000000000 ␣
↪27575190
035e1c00620000000000 ... 1f97e1191304710000080040200008000004002010000400000000000 ␣
↪27575192
075e1c00020000000000 ... 1f97e1191305710100000200680000000000010034000000000000000 ␣
↪27575198
075e1c00024000000000 ... 1f97e1191304710000000000020000000000000001000000000000000 ␣
↪27575208
```

How does this work? Each PubChem record contains the precomputed CACTVS substructure keys in the PUBCHEM_CACTVS_SUBSKEYS tag. Here's what it looks like for record 27575190, which is the first record in Compound_027575001_027600000.sdf.gz:

```
> <PUBCHEM_CACTVS_SUBSKEYS>
AAADceB6OABAEAAAAAAAAAAAAAAAAAAAAAAwYAAAAAAAAAABQAAAHgRQAAABrAil2AKyyYLABAqIAiXS
WHLCAAAlChQIiBlAbOgKJjLgtZ2HMQhk1AH465eYyCCOAAAgQAAEgAAAAECAAAkAAAAAAAAAA==
```

The `--pubchem` flag tells *sdf2fps* to get the value of that tag and decode it to get the fingerprint. It also adds a few metadata fields to the fingerprint file header.

The order of the FPS fingerprints are the same as the order of the corresponding record in the SDF. You can see that in the output, where 27575190 is the first record in the FPS fingerprints.

If you store records in an SD file then you almost certainly don't use the same fingerprint encoding as PubChem. *sdf2fps* can decode from a number of encodings, like hex and base64. Use `--help` to see the list of available decoders.

k-nearest neighbor search

In this section you'll learn how to search a fingerprint file to find the k-nearest neighbors. You will need the FPS fingerprint files generated in *Generate fingerprint files from PubChem SD tags* but you do not need a chemistry toolkit.

We'll use the pubchem_queries.fps as the queries for a k=2 nearest neighor similarity search of the target file puchem_targets.gps:

```
simsearch -k 2 -q pubchem_queries.fps pubchem_targets.fps
```

That's all! You should get output which starts:

```
#Simsearch/1
#num_bits=881
#type=Tanimoto k=2 threshold=0.0
#software=chemfp/3.1
#queries=pubchem_queries.fps
#targets=pubchem_targets.fps
#query_sources=Compound_027575001_027600000.sdf.gz
#target_sources=Compound_014550001_014575000.sdf.gz
2   27575190        14555201        0.7236  14566941        0.7105
2   27575192        14555203        0.7158  14555201        0.7114
2   27575198        14555201        0.7286  14569555        0.7259
2   27575208        14555201        0.7701  14566941        0.7584
```

Here's how to interpret the output. The lines starting with '#' are header lines. It contains metadata information describing that this is a similarity search report. You can see the search parameters, the name of the tool which did the search, and the filenames which went into the search.

After the '#' header lines come the search results, with one result per line. There are in the same order as the query fingerprints. Each result line contains tab-delimited columns. The first column is the number of hits. The second column is the query identifier used. The remaining columns contain the hit data, with alternating target id and its score.

For example, the first result line contains the 2 hits for the query 27575190. The first hit is the target id 14555201 with score 0.7236 and the second hit is 14566941 with score 0.7105. Since this is a k-nearest neighor search, the hits are sorted by score, starting with the highest score. Do be aware that ties are broken arbitrarily.

Threshold search

In this section you'll learn how to search a fingerprint file to find all of the neighbors at or above a given threshold. You will need the FPS fingerprint files generated in *Generate fingerprint files from PubChem SD tags* but you do not need a chemistry toolkit.

Let's do a threshold search and find all hits which are at least 0.738 similar to the queries:

```
simsearch --threshold 0.738 -q pubchem_queries.fps pubchem_targets.fps
```

The first 14 lines of output from this are:

```
#Simsearch/1
#num_bits=881
```

```
#type=Tanimoto k=all threshold=0.738
#software=chemfp/3.1
#queries=pubchem_queries.fps
#targets=pubchem_targets.fps
#query_sources=Compound_027575001_027600000.sdf.gz
#target_sources=Compound_014550001_014575000.sdf.gz
0   27575190
0   27575192
0   27575198
3   27575208        14566941        0.7584  14566938        0.7542  14555201        0.
↪7701
3   27575221        14566941        0.7473  14566938        0.7432  14555201        0.
↪7592
3   27575223        14566941        0.7473  14566938        0.7432  14555201        0.
↪7592
```

Take a look at the last line, which contains the 3 hits for the query id 27575223. As before, the hit information alternates between the target ids and the target scores, but unlike the k-nearest search, the threshold search hits are not in a particular order. You can see that here with the scores 0.7473, 0.7432, 0.7592, which are in neither increasing nor decreasing order.

Combined k-nearest and threshold search

In this section you'll learn how to search a fingerprint file to find the k-nearest neighbors, where all of the hits must be at or above given threshold. You will need the fingerprint files generated in *Generate fingerprint files from PubChem SD tags* but you do not need a chemistry toolkit.

You can combine the `-k` and `--threshold` queries to find the k-nearest neighbors which are all above a given threshold:

```
simsearch -k 3 --threshold 0.7 -q pubchem_queries.fps pubchem_targets.fps
```

This find the nearest 3 structures, which all must be at least 0.7 similar to the query fingerprint. The output from the above starts:

```
#Simsearch/1
#num_bits=881
#type=Tanimoto k=3 threshold=0.7
#software=chemfp/3.1
#queries=pubchem_queries.fps
#targets=pubchem_targets.fps
#query_sources=Compound_027575001_027600000.sdf.gz
#target_sources=Compound_014550001_014575000.sdf.gz
3   27575190        14555201        0.7236  14566941        0.7105  14566938        0.
↪7068
2   27575192        14555203        0.7158  14555201        0.7114
3   27575198        14555201        0.7286  14569555        0.7259  14553070        0.
↪7065
3   27575208        14555201        0.7701  14566941        0.7584  14566938        0.
↪7542
3   27575221        14555201        0.7592  14566941        0.7473  14566938        0.
↪7432
3   27575223        14555201        0.7592  14566941        0.7473  14566938        0.
↪7432
2   27575240        14555201        0.7150  14566941        0.7016
```

```
2   27575250        14555203        0.7128  14555201        0.7085
3   27575257        14572463        0.7468  14563588        0.7250  14561245        0.
↪7219
```

The output format is identical to the previous two search examples, and because this is a k-nearest search, the hits are sorted from highest score to lowest.

NxN (self-similar) searches

In this section you'll learn how to use the same fingerprints as both the queries and targets, that is, a self-similarity search. You will need the pubchem_queries.fps fingerprint file generated in *Generate fingerprint files from PubChem SD tags* but you do not need a chemistry toolkit.

Use the `--NxN` option if you want to use the same set of fingerprints as both the queries and targets. Using the pubchem_queries.fps from the previous sections:

```
simsearch -k 3 --threshold 0.7 --NxN pubchem_queries.fps
```

This code is very fast because there are so few fingerprints. For larger files the `--NxN` will be about twice as fast and use half as much memory compared to:

```
simsearch -k 3 --threshold 0.7 -q pubchem_queries.fps pubchem_queries.fps
```

In addition, the `--NxN` option excludes matching a fingerprint to itself (the diagonal term).

Using a toolkit to process the ChEBI dataset

In this section you'll learn how to create a fingerprint file from a structure file. The structure processing and fingerprint generation are done with a third-party chemisty toolkit. chemfp supports Open Babel, OpenEye, and RDKit. (OpenEye users please note that you will need an OEGraphSim license to use the OpenEye-specific fingerprinters.)

We'll work with data from ChEBI, which are "Chemical Entities of Biological Interest". They distribute their structures in several formats, including as an SD file. For this section, download the "lite" version from ftp://ftp.ebi.ac.uk/pub/databases/chebi/SDF/ChEBI_lite.sdf.gz . It contains the same structure data as the complete version but many fewer tag data fields. For ChEBI 155 this file contains 95,955 records and the compressed file is 28MB.

Unlike the PubChem data set, the ChEBI data set does not contain fingerprints so we'll need to generate them using a toolkit.

ChEBI record titles don't contain the id

Strangely, the ChEBI dataset does not use the title line of the SD file to store the record id. A simple examination shows that 47,376 of the title lines are empty, 39,615 have the title "null", 4,499 have the title " ", 2,033 have the title "ChEBI", 45 of them are labeled "Structure #1", and the others are usually compound names.

(I've asked ChEBI to fix this, to no success. Perhaps you have more influence?)

Instead, the record id is stored as value of the "ChEBI ID" tag, which looks like:

```
> <ChEBI ID>
CHEBI:776
```

By default the toolkit-based fingerprint generation tools use the title as the identifier, and print a warning and skip the record if the identifier is missing. Here's an example with *rdkit2fps*:

```
% rdkit2fps ChEBI_lite.sdf.gz
#FPS1
#num_bits=2048
#type=RDKit-Fingerprint/2 minPath=1 maxPath=7 fpSize=2048 nBitsPerHash=2 useHs=1
#software=RDKit/2017.09.1.dev1 chemfp/3.1
#source=ChEBI_lite.sdf.gz
#date=2017-09-14T21:17:44
ERROR: Missing title in SD record, file 'ChEBI_lite.sdf.gz', line 1, record #1.␣
↪Skipping.
ERROR: Missing title in SD record, file 'ChEBI_lite.sdf.gz', line 62, record #2.␣
↪Skipping.
ERROR: Missing title in SD record, file 'ChEBI_lite.sdf.gz', line 100, record #3.␣
↪Skipping.
ERROR: Missing title in SD record, file 'ChEBI_lite.sdf.gz', line 135, record #4.␣
↪Skipping.
ERROR: Missing title in SD record, file 'ChEBI_lite.sdf.gz', line 201, record #5.␣
↪Skipping.
ERROR: Missing title in SD record, file 'ChEBI_lite.sdf.gz', line 236, record #6.␣
↪Skipping.
[23:17:44]  S group MUL ignored on line 103
ERROR: Missing title in SD record, file 'ChEBI_lite.sdf.gz', line 264, record #7.␣
↪Skipping.
[23:17:44]  Unhandled CTAB feature: S group SRU on line: 31. Molecule skipped.
ERROR: Missing title in SD record, file 'ChEBI_lite.sdf.gz', line 435, record #9.␣
↪Skipping.
ERROR: Missing title in SD record, file 'ChEBI_lite.sdf.gz', line 519, record #10.␣
↪Skipping.
ERROR: Missing title in SD record, file 'ChEBI_lite.sdf.gz', line 581, record #11.␣
↪Skipping.
031087be231150242e714400920000a193c1080c02858a1116a68100a588063428404052530040 80c8cc3c4811
4101b25081a10c025e634c08a1c00088102c0400121040a2080505188a9c0a150000028211219c1001000981c4
804417180aca040140850018018221071 6db1580708a0b8a08028205328544112 00c1101040404001118600d0a
518402385dc00011290602205a070480c148f240421000c321801922c7808740cd0b10ea4c40000403dc180121
94d8d120020150b3d00043a24370000201042881d15018c0e0901442881d68604c4a83808110c772a824051948
003c80136060022104001 0e20418381668404b0424ec130f05a090c94960e0      ChEBI
000080000000000000000028800000000000000020000004008000000000000002000400000020 00c000000
000000000080080000002004001000000000000000100000400001000000000000008000000000000100
000080100200000001000000400004c000000000000008000040000000011020000002000040000001003 00
080000000000000000000000000000000000082000040400000080000040000020 0c000080400000000000000000
200101008000000000000000000202000002008000000000000002000000000008000400000000000000100
400001000200800000010003002800000020020000000000000000000000000      ChEBI
    ....
```

That output contains only two fingerprint records, both with the id "ChEBI". The other records had no title and were skipped, with a message sent to stderr describing the problem and the location of the record containing the problem.

(If the first 100 records have no identifiers then the command-line tools will exit even if `--errors` is ignore. This is a safety mechanism. Let me know if it's a problem.)

Instead, use the `--id-tag` option to specify of the name of the data tag containing the id. For this data set you'll need to write it as:

```
--id-tag "ChEBI ID"
```

The quotes are important because of the space in the tag name.

Here's what that looks like:

```
% rdkit2fps --id-tag "ChEBI ID" ChEBI_lite.sdf.gz  | fold | head -20
[23:26:39]  S group MUL ignored on line 103
[23:26:39]  Unhandled CTAB feature: S group SRU on line: 31. Molecule skipped.
#FPS1
#num_bits=2048
#type=RDKit-Fingerprint/2 minPath=1 maxPath=7 fpSize=2048 nBitsPerHash=2 useHs=1
#software=RDKit/2017.09.1.dev1 chemfp/3.1
#source=ChEBI_lite.sdf.gz
#date=2017-09-14T21:26:39
10208220141258c184490038b4124609db0030024a0765883c62c9e1288a1dc224de62f445743b8b
30ad542718468104d521a214227b29ba3822fbf20e15491802a051532cd10d902c39b02b51648981
9c87eb41142811026d510a890a711cb02f2090ddacd990c5240cc282090640103d0a0a8b460184f5
11114e2a80602008045298045323 13bb03912d5e2857a6028960189e370100052c63474748a1c000
8079f49c484ca04c0d0bcb2c64b72401042a1f82002b097e852830e5898302021a1203e412064814
a598741c014e9210bc30ab180f0162029d4c446aa01c34850071e4ff037a60e732fd85014344f82a
344aa98398654481b003a84f201f518f    CHEBI:90
0000000080200412008000008000004000010100022008000400002000020100020006000800001
0100010008000100001000000200220000020000008000000400002100000000080000004401000
8020002080020000200000140002206400000424481000000000080000a80012002020004198002
0008020002002012004020300100080201010002421100000440000000010020000300000100 100
0100021000a200601080002a000020200480040300008840840000080000020402000108000000000
2000010022000800002000020001400020800100025040000000200a08024400006000800000802
8100c801108000000041c00200800002    CHEBI:165
```

In addition to "ChEBI ID" there's also a "ChEBI Name" tag which includes data values like "tropic acid" and "(+)-guaia-6,9-diene". Every ChEBI record has a unique name so the names could also be used as the primary identifier instead of its id.

The FPS fingerprint file format allows identifiers with a space, or comma, or anything other tab, newline, and a couple of other bytes, so it's no problem using those names directly.

To use the ChEBI Name as the primary chemfp identifier, specify:

```
--id-tag "ChEBI Name"
```

Generate fingerprints with Open Babel

If you have the Open Babel Python library installed then you can use *ob2fps* to generate fingerprints:

```
ob2fps --id-tag "ChEBI ID" ChEBI_lite.sdf.gz -o ob_chebi.fps
```

This takes just under 3 minutes on my 7 year old desktop to process all of the records.

The default generates FP2 fingerprints, so the above is the same as:

```
ob2fps --FP2 --id-tag "ChEBI ID" ChEBI_lite.sdf.gz -o ob_chebi.fps
```

ob2fps can generate several other types of fingerprints. (Use `--help` for a list.) For example, to generate the Open Babel implementation of the MACCS definition specify:

```
ob2fps --MACCS --id-tag "ChEBI ID" ChEBI_lite.sdf.gz -o chebi_maccs.fps
```

Generate fingerprints with OpenEye

If you have the OEChem Python library installed, with licenses for OEChem and OEGraphSim, then you can use *oe2fps* to generate fingerprints:

```
oe2fps --id-tag "ChEBI ID" ChEBI_lite.sdf.gz -o oe_chebi.fps
```

This takes about 40 seconds on my desktop and generates a number of warnings like "Stereochemistry corrected on atom number 17 of", "Unsupported Sgroup information ignored", and "Invalid stereochemistry specified for atom number 9 of". Normally the record title comes after the "... of", but the title is blank for most of the records.

OEChem could not parse 7 of the 95,955 records. I looked at the failing records and noticed that all of them had 0 atoms and 0 bonds.

The default settings generate OEGraphSim path fingerprint with the values:

```
numbits=4096 minbonds=0 maxbonds=5
   atype=Arom|AtmNum|Chiral|EqHalo|FCharge|HvyDeg|Hyb btype=Order|Chiral
```

Each of these can be changed through command-line options. Use `--help` for details.

oe2fps can generate several other types of fingerprints. For example, to generate the OpenEye implementation of the MACCS definition specify:

```
oe2fps --maccs166 --id-tag "ChEBI ID" ChEBI_lite.sdf.gz -o chebi_maccs.fps
```

Use `--help` for a list of available oe2fps fingerprints or to see more configuration details.

Generate fingerprints with RDKit

If you have the RDKit Python library installed then you can use *rdkit2fps* to generate fingerprints. Based on the previous examples you probably guessed that the command-line is:

```
rdkit2fps --id-tag "ChEBI ID" ChEBI_lite.sdf.gz -o rdkit_chebi.fps
```

This takes 5.5 minutes on my desktop, and RDKit did not generate fingerprints for 1,101 of the 95,955 records. RDKit logs warning and error messages to stderr. They look like:

```
[23:29:49] Explicit valence for atom # 6 N, 4, is greater than permitted
[23:29:49]

****
Post-condition Violation
Element '.' not found
Violation occurred on line 90 in file /Users/dalke/cvses/rdkit/Code/GraphMol/
↪PeriodicTable.h
Failed Expression: anum > -1
****

[23:29:49]  Unhandled CTAB feature: S group SRU on line: 52. Molecule skipped.
```

For example, RDKit is careful to check that structures make chemical sense. It rejects 4-valent nitrogen and refuses to process that those structures, which is the reason for the first line of that output.

The default generates RDKit's path fingerprints with parameters:

```
minPath=1 maxPath=7 fpSize=2048 nBitsPerHash=2 useHs=1
```

Each of those can be changed through command-line options. See rdkit2fps `--help` for details, where you'll also see a list of the other available fingerprint types.

For example, to generate the RDKit implementation of the MACCS definition use:

```
rdkit2fps --maccs166 --id-tag "ChEBI ID" ChEBI_lite.sdf.gz -o chebi_maccs.fps
```

while the following generates the Morgan/circular fingerprint with radius 3:

```
rdkit2fps --morgan --radius 3 --id-tag "ChEBI ID" ChEBI_lite.sdf.gz
```

Alternate error handlers

In this section you'll learn how to change the error handler for rdkit2fps using the `--errors` option.

By default the "<toolkit>2fps" programs "ignore" structures which could not be parsed into a molecule option. There are two other options. They can "report" more information about the failure case and keep on processing, or they can be "strict" and exit after reporting the error.

This is configured with the `--errors` option.

Here's the rdkit2fps output using `--errors report`:

```
[00:52:39]  S group MUL ignored on line 103
[00:52:39]  Unhandled CTAB feature: S group SRU on line: 36. Molecule skipped.
ERROR: Could not parse molecule block, file 'ChEBI_lite.sdf.gz', line 12036, record
↪#179. Skipping.
[00:52:39] Explicit valence for atom # 12 N, 4, is greater than permitted
ERROR: Could not parse molecule block, file 'ChEBI_lite.sdf.gz', line 16213, record
↪#265. Skipping.
```

The first two lines come from RDKit. The third line is from chemfp, reporting which record could not be parsed. (The record starts at line 12036 of the file and the SRU is on line 36 of the record, so the SRU is at line 12072.) The fourth line is another RDKit error message, and the last line is another chemfp error message.

Here's the rdkit2fps output using `--errors strict`:

```
[00:54:30]  S group MUL ignored on line 103
[00:54:30]  Unhandled CTAB feature: S group SRU on line: 36. Molecule skipped.
ERROR: Could not parse molecule block, file 'ChEBI_lite.sdf.gz', line 12036, record
↪#179. Exiting.
```

Because this is strict mode, processing exits at the first failure.

The ob2fps and oe2fps tools implement the `--errors` option, but they aren't as useful as rdkit2fps because the underlying APIs don't give useful feedback to chemfp about which records failed. For example, the standard OEChem file reader automatically skips records that it cannot parse. Chemfp can't report anything when it doesn't know there was a failure.

The default error handler in chemfp 1.1 was "strict". In practice this proved more annoying than useful because most people want to skip the records which could not be processed. They would then contact me asking what was wrong, or doing some pre-processing to remove the failure cases.

One of the few times when it is useful is for records which contain no identifier. When I changed the default from "strict" to "ignore" and tried to process ChEBI, I was confused at first about why the output file was so small. Then I realized that it's because the many records without a title were skipped, and there was no feedback about skipping those records.

I changed the code so missing identifiers are always reported, even if the error setting is "ignore". Missing identifiers will still stop processing if the error setting is "strict".

chemfp's two cross-toolkit substructure fingerprints

In this section you'll learn how to generate the two substructure-based fingerprints which come as part of chemfp. These are based on cross-toolkit SMARTS pattern definitions and can be used with Open Babel, OpenEye, and RDKit. (For OpenEye users, these fingerprints use the base OEChem library but do not use the separately licensed OEGraphSim library.)

chemfp implements two platform-independent fingerprints where were originally designed for substructure filters but which are also used for similarity searches. One is based on the 166-bit MACCS implementation in RDKit and the other comes from the 881-bit PubChem/CACTVS substructure fingerprints.

The chemfp MACCS definition is called "rdmaccs" because it closely derives from the MACCS SMARTS patterns used in RDKit. (These pattern definitions are also used in Open Babel and the CDK, while OpenEye has a completely independent implementation.)

Here are example of the respective rdmaccs fingerprint for phenol using each of the toolkits.

Open Babel:

```
% echo "c1ccccc1O phenol" | ob2fps --in smi --rdmaccs
#FPS1
#num_bits=166
#type=RDMACCS-OpenBabel/2
#software=OpenBabel/2.4.1 chemfp/3.1
#date=2017-09-09T00:40:48
00000000000000000000000000000140004480101e  phenol
```

OpenEye:

```
#FPS1
#num_bits=166
#type=RDMACCS-OpenEye/2
#software=OEChem/2.1.3 (20170828) chemfp/3.1
#date=2017-09-09T00:41:21
00000000000000000000000000000140004480101e  phenol
```

RDKit:

```
#FPS1
#num_bits=166
#type=RDMACCS-RDKit/2
#software=RDKit/2017.09.1 chemfp/3.1
#date=2017-09-09T00:42:32
00000000000000000000000000000140004480101e  phenol
```

For more complex molecules it's possible that different toolkits produce different fingerprint rdmaccs, even though the toolkits use the same SMARTS definitions. Each toolkit has a different understanding of chemistry. The most notable is the different definition of aromaticity, so the bit for "two or more aromatic rings" will be toolkit dependent.

substruct fingerprints

chemp also includes a "substruct" substructure fingerprint. This is an 881 bit fingerprint derived from the PubChem/CACTVS substructure keys. They do not match the CACTVS fingerprints exactly, in part due to differences in ring perception. Some of the substruct bits will always be 0. With that caution in mind, if you want to try them out, use the `--substruct` option.

The term "substruct" is a horribly generic name. If you can think of a better one then let me know. Until chemfp 3.0 I said these fingerprints were "experimental", in that I hadn't fully validated them against PubChem/CACTVS and could not tell you the error rate. I still haven't done that.

What's changed is that I've found out over the years that people are using the substruct fingerprints, even without full validatation. That surprised me, but use is its own form of validation. I still would like to validate the fingerprints, but it's slow, tedious work which I am not really interested in doing. Nor does it earn me any money. Plus, if the validation does lead to any changes, it's easy to simply change the version number.

Generate binary FPB files from a structure file

In this section you'll learn how to generate an FPB file instead of an FPS file. You will need the the ChEBI file from *Using a toolkit to process the ChEBI dataset* and a chemistry toolkit. The FPB format was introduced with chemfp-2.0.

The FPB format was designed so the fingerprints can be memory-mapped directly to chemfp's internal data structures. This makes it very fast to load, but unlike the FPS format, it's not so easy to write with your own code. You should think of the FPB format as an binary application format, for chemfp-based tools, while the FPS format is a text-based format for data exchange between diverse programs.

The easiest way to generate an FPB file from the command line is to use the ".fpb" extension instead of ".fps" or ".fps.gz". Here are examples using each of the toolkits.

Open Babel:

```
% ob2fps --id-tag "ChEBI ID" ChEBI_lite.sdf.gz -o ob_chebi.fpb
```

OpenEye:

```
% oe2fps --id-tag "ChEBI ID" ChEBI_lite.sdf.gz -o oe_chebi.fpb
```

RDKit:

```
% rdkit2fps --id-tag "ChEBI ID" ChEBI_lite.sdf.gz -o rdkit_chebi.fpb
```

The binary format isn't human-readable. Use *fpcat command-line options* to see what's inside:

```
% fpcat oe_chebi.fpb | head -8
#FPS1
#num_bits=4096
#type=OpenEye-Path/2 numbits=4096 minbonds=0 maxbonds=5␣
↪atype=Arom|AtmNum|Chiral|EqHalo|FCharge|HvyDeg|Hyb btype=Order|Chiral
#software=OEGraphSim/2.3.1 (20170828) chemfp/3.1
000 ... many zeros ... 000  CHEBI:15378
000 ... many zeros ... 000  CHEBI:16042
000 ... many zeros ... 000  CHEBI:17792
000 ... many zeros ... 000  CHEBI:18140
           ....
182 ... hex values ... c0c  CHEBI:60493
```

By default the fingerprints are ordered from smallest popcount to largest, which you can see in the output. A pre-ordered index is faster to search because the target popcounts are pre-computed and because it enables sublinear search.

If you want to preserve the input order then you'll need to pipe the FPS output to *fpcat* and use its `--preserve-order` flag. See the next section for an example.

Convert between FPS and FPB formats

In this section you'll learn how to convert an FPS file into an FPB file and back, and you'll learn how to control the fingerprint ordering. You will need the FPS files generated in *Generate fingerprint files from PubChem SD tags* but you do not need a chemistry toolkit. The FPB format was introduced with chemfp-2.0.

If you already have an FPS file then you can convert it directly into an FPB file, and without using a chemistry toolkit. The *fpcat* program converts from one format to the other.

In an earlier section I generated the files pubchem_queries.fps and pubchem_targets.fps . I'll convert each to FPB format:

```
% fpcat pubchem_targets.fps -o pubchem_targets.fpb
% fpcat pubchem_queries.fps -o pubchem_queries.fpb
```

The FPB format is a binary format which is difficult to read directly. The easiest way to see what's inside is to use fpcat. If you don't specify an output filename then it sends the results to stdout in FPS format:

```
% fpcat pubchem_queries.fpb | head -5
#FPS1
#num_bits=881
#type=CACTVS-E_SCREEN/1.0 extended=2
#software=CACTVS/unknown
071e0c0000000000000000000000000080c16030000c060000000000000000000005800000000
00f020010100400000000001000010800000000000000000000800000000004800000000000000
0000000080901103f10100000000000000100200000400800000100000000      27581954
```

The keen-eyed reader might have noticed that the conversion does not have a "source" or "date" field. I haven't figured out if this is a bug. Should I keep the original date and structure file source, or use the current date and FPS file source? Let me know if this is important to you.

By default fpcat reorders the fingerprints in the FPB file by population count. This improves the similarity search performance, but it means that the order of the FPB file is likely different than the original FPS format. You can get a sense of this by looking at the first fingerprint in the original pubchem_queries.fps file:

```
% grep -v # pubchem_queries.fps  | head -1
075e1c0002080000000000000000000000000000000c060000000000000800200007820 0a0000
803510a51b404d93410320501140a44b1a4e430000a450281011980236175 0644c07adb9e18c1026
2b801fd7e919130471000004020020010000000201009000000000000000      27575190
```

If you want the FPB file to store the fingerprints in input order instead of the popcount order needed for optimized similarity search, then use the `--preserve-order` flag:

```
% fpcat pubchem_queries.fps --preserve-order -o input_order.fpb
% fpcat input_order.fpb | grep -v # | head -1
075e1c0002080000000000000000000000000000000c060000000000000800200007820 0a0000
803510a51b404d93410320501140a44b1a4e430000a450281011980236175 0644c07adb9e18c1026
2b801fd7e919130471000004020020010000000201009000000000000000      27575190
```

On the flip side, fpcat by default preserves the input order when it creates FPS output. If you instead want to the output FPS file to be in popcount order then use the `--reorder` flag:

```
% fpcat --reorder pubchem_queries.fps | grep -v # | head -1
071e0c0000000000000000000000000080c16030000c060000000000000000000005800000000
00f020010100400000000001000010800000000000000000000800000000004800000000000000
0000000080901103f10100000000000000100200000400800000100000000      27581954
```

Specify the fpcat output format

In this section you'll learn how to specify the output format for fpcat using a command-line option instead of the filename extension. You will need the pubchem_queries.fpb file from *Generate fingerprint files from PubChem SD tags*.

If you do not specify an output filename then fpcat will output the fingerprints in FPS format to stdout. If you specify a filename then by default it will look at the extension to determine if the output should be an FPB (".fpb"), FPS (".fps"), or gzip compressed FPS (".fps.gz") file. The FPS format is used for unrecognized extensions.

In a few rare cases you may want to use a format which doesn't match the default. To be honest, the examples I can think of aren't that realistic, but let's suppose you want to output the contents of an FPB file to stdout in gzip'ed FPS format, and count the number of bytes in compressed output. I'll use the use the –out flag to change the format to 'fps.gz' from the default of 'fps', then compare the resulting size with the uncompressed form:

```
% fpcat pubchem_queries.fpb --out fps.gz | wc -c
11930
% fpcat pubchem_queries.fpb --out fps | wc -c
89170
```

It's not that useful because you could pipe the uncompressed output to gzip, which is also likely faster:

```
% fpcat pubchem_queries.fpb --out fps | gzip -c -9 | wc -c
11921
```

By the way, it is not possible to write an FPB file to stdout. In fact, the output file must be seek-able, which means it can't be a named pipe either.

Similarity search with the FPB format

In this section you'll learn how to do a similarity search using an FPB as the target. You will need the FPB files from *Generate fingerprint files from PubChem SD tags* but you do not need a chemistry toolkit.

Simsearch, like all of the tools starting with chemfp-2.0, understands both FPS and FPB files:

```
% simsearch -k 3 --threshold 0.7 -q pubchem_queries.fpb pubchem_targets.fpb | head
#Simsearch/1
#num_bits=881
#type=Tanimoto k=3 threshold=0.7
#software=chemfp/3.1
#queries=pubchem_queries.fpb
#targets=pubchem_targets.fpb
3   27581954        14565747        0.7833  14563541        0.7333  14573233        0.
↪7258
3   27581957        14565747        0.7833  14563541        0.7333  14573233        0.
↪7258
3   27580389        14568366        0.8468  14568369        0.8393  14560737        0.
↪8374
2   27584917        14563095        0.7795  14563096        0.7795
```

By default simsearch uses the query and target filename extensions to figure out if the file is in FPS or FPB format.

If you don't want it to auto-detect the format then use the `--query-format` and `--target-format` options to tell it the format to use. The values can be one of "fps", "fps.gz" and "fpb".

Converting large data sets to FPB format

In this section you'll learn how to generate an FPB file on computers with relatively limited memory. To be realistic, this example uses the complete PubChem data set, and extracts the CACTVS/PubChem fingerprints which are in each record. You do not need a chemistry toolkit for this section.

The most direct way to extract the PubChem fingerprints from a PubChem distribution is to use *sdf2fps*:

```
sdf2fps --pubchem pubchem/Compound_*.sdf.gz -o pubchem.fpb
```

This uses the default FPB writer options, which stores all of the fingerprints in memory, sorts them, and saves the result to the output file. This may use about 2-3 times as much memory as the final FPB output size, which is a bit unfortunate if you want to generate a 7 GB FPB file on a 12 GB machine.

(Note: see *the next section* for a two-stage solution that lets you parallelize fingerprint generation.)

The "*2fps" command-line tools do not have a way to change the default writer options, although *fpcat* does. The `--max-spool-size` option sets a rough upper bound to the amount of memory to use. When enabled, the writer breaks the input into parts and creates a temporary FPB file for each part. At the end, it merges the sorted data from the temporary FPB files to get the final FPB file. Note that the specified spool size is only approximate and is not a hard limit on the maximum amount of memory to use. You may need to experiment a bit if you have tight constraints.

The value must be a size in bytes, though suffixes like M or MB for megabyte and T or TB for terabyte are also allowed. These are in base-10 units, so 1 MB = 1,000,000 bytes. Spaces are not allowed between the number and the suffix, so "200MB" is okay but "200 MB" is not. The size must be at least 20 MB.

Here is an example of how to convert the CACTVS fingerprints from all of PubChem to an FPB file, using a relatively small limit of 200 MB:

```
sdf2fps --pubchem pubchem/Compound_*.sdf.gz | fpcat --max-spool-size 200MB -o pubchem.
↪fpb
```

This will take a while! The sdf2fps alone takes almost 45 minutes on my desktop, of which 50% of the time was to decompress the files.

The temporary files will be placed under the appropriate temporary directory for your operating system. If that disk isn't large enough for the intermediate files then use the `--tmpdir` option of fpcat to specify an alternate directory:

```
fpcat --max-spool-size 1GB pubchem.fps -o pubchem.fpb
```

Another option is to specify the directory location using the TMPDIR, TEMP, or TMP environment variables, which are resolved in that order. The details are described in the Python documentation for tempfile.tempdir.

Generate fingerprints in parallel and merge to FPB format

In this section you'll learn how to merge multiple sorted fingerprints into a single FPB file.

The previous section used a single shell command to extract the PubChem/CACTVS fingerprints from PubChem and generate an FPB file. This is easy to write and understand, but more complex versions may be more appropriate.

For one, I have four cores on my desktop computer, and I want to use them to process the PubChem files in parallel. The previous section was only single threaded.

I have all my PubChem files in `~/pubchem/`. For each "Compound_*.sdf.gz" file in that directory I want to extract the CACTVS/PubChem fingerprints and create an intermediate FPS file in the local directory. That's equivalent to running the following commands:

```
sdf2fps --pubchem ~/pubchem/Compound_000000001_000025000.sdf.gz \\
     -o Compound_000000001_000025000.fps
sdf2fps --pubchem ~/pubchem/Compound_000025001_000050000.sdf.gz \\
     -o Compound_000025001_000050000.fps
 ... 2146 more lines ...
```

except that I want to run four at a time.

This is what GNU Parallel was designed for. It's a command-line tool which can parallelize the exection of other command-lines.

I'll start by explaining the core command-line substitution pattern:

```
sdf2fps --pubchem {} -o {/..}.fps'
```

The `{}` will be replaced with a filename, and `{/..}` will be replaced with the base filename, without the directory path prefix or the two suffixes. That is, when `{}` is "/Users/dalke/pubchem/Compound_000000001_000025000.sdf.gz" then `{/..}` will be "Compound_000000001_000025000".

Since I want to generate an FPS file, I added the ".fps" as a suffix to the second substitution parameter.

I then tell GNU parallel which command-line to use, along with a few other parameters. Here's the full line, which I split over two lines to make it more readable:

```
parallel --plus --no-notice --bar 'sdf2fps --pubchem {}
   -o {/..}.fps' ::: ~/pubchem/Compound_*.sdf.gz
```

The `--plus` tells GNU parallel to recognize an expanded set of replacement strings. ("{/..}" is not part of the standard set of patterns.)

The `--no-notice` tells it to not display the message about citing GNU parallel in scientific papers.

The `--bar` enables a progress bar, which looks like this:

```
26% 763:2148=1696s /Users/dalke/pubchem/Compound_019150001_019175000.sdf.gz
```

It's 26% through processing the filenames, which is file 763 out of 2148, and there's an estimated 1696 seconds remaining.

Finally, the ":::" indicates that the remaining options are the list of parameters to pass to the command-line template for parallelization.

After about 30 minutes, I now have a large number of FPS files, which I want to merge into a single FPB file. I'll use *fpcat*:

```
fpcat --max-spool-size 2GB Compound*.fps -o pubchem.fpb
```

This took about 15 minutes. (It's a bit odd that the overall performance wasn't that much better than the single-threaded code. It would probably be more clear with compute-intenstive fingerprints, instead of simple text extraction from an SD tag.)

Note: I started this section as an example of when to use the `--merge` option to fpcat. When the fingerprints are in popcount order then popcount sorted output is a merge sort of the inputs. This doesn't need RAM or temporary disk space for an intermediate sort. My thought was to save the intermediate fingerprints in FPB format instead of FPS, which has a side-effect of sorting the fingerprints. Then I could simply merge the results.

I did this, and ran into two problems. There are 2912 files, and fpcat will open all of them in order to do the parallel merge. I ran out of file descriptors, and had to increase the limit to 6000 (3000 is too small) before it would work. In the future I'll have to implement some sort of multi-layer merge for when there are too many files. However, even

with 6000 available descriptors, iterating over the FPB-backed FingerprintArena proved to be rather slow, and I'm not yet sure way. I think it's simply that I didn't design that code for fast iteration.

Take home message? Use FPB files for now only as the last file format in your pipeline.

Help for the command-line tools

The chemfp command-line tools are:

- *fpcat* - merge multiple fingerprint files into one
- *ob2fps* - use Open Babel to generate fingerprints
- *oe2fps* - use OEChem/OEGraphSim to generate fingerprints
- *rdkit2fps* - use RDKit to generate fingerprints
- *sdf2fps* - extract fingerprints from an SD file
- *simsearch* - search a fingerprint file for similar fingerprints

fpcat command-line options

The following comes from `fpcat --help`:

```
usage: fpcat [-h] [--in FORMAT] [--merge] [-o FILENAME] [--out FORMAT]
             [--reorder] [--preserve-order] [--alignment N]
             [--show-progress] [--max-spool-size SIZE] [--tmpdir DIRNAME]
             [--version]
             [filename [filename ...]]

Combine multiple fingerprint files into a single file.

positional arguments:
  filename              input fingerprint filenames (default: use stdin)

optional arguments:
  -h, --help            show this help message and exit
  --in FORMAT           input fingerprint format. One of fps, fps.gz, or fpb.
                        (default guesses from filename or is fps)
  --merge               assume the input fingerprint files are in popcount
                        order and do a merge sort
  -o FILENAME, --output FILENAME
                        save the fingerprints to FILENAME (default=stdout)
  --out FORMAT          output fingerprint format. One of fps, fps.gz, or fpb.
                        (default guesses from output filename, or is 'fps')
  --reorder             reorder the output fingerprints by popcount (default
                        for FPB output)
  --preserve-order      save the output fingerprints in the same order as the
                        input (default for FPS output)
  --alignment N         alignment size when saving a FPB file (default=8)
  --show-progress       show progress
  --max-spool-size SIZE
                        use temporary files for extra storage space for huge
                        FPB files (default uses RAM)
  --tmpdir DIRNAME      directory for the temporary files (default uses the
                        system temp directory)
  --version             show program's version number and exit
```

```
Examples:

fpcat can be used to convert between FPS and FPB formats. This is
handy if you want to see what's inside of an FPB file:

    fpcat fingerprints.fpb

You can use also use fpcat to make an FPB file from an FPS file:

    fpcat fingerprints.fps -o fingerprints.fpb

You might have generated a set of FPS file which you want to merge
into a single FPB. (For example, you might have used GNU parallel to
generate FPS files for each of the PubChem files, which you want to
merge into a single file.):

    fpcat Compound_*.fps -o pubchem.fpb

By default the FPB format sorts the fingerprints by popcount. (Use
--preserve-order if you really want to preserve the input order.)  The
sort overhead for PubChem uses about 10 GB of RAM. If you don't have
that much memory then ask fpcat to use less memory:

    fpcat --max-spool-size 1GB Compound_*.fps -o pubchem.fpb

This will use about 2 GB of RAM and the --tmpdir for the rest. (Yes,
it would be nice if I could get those two memory size numbers to
match.)

The --merge option is experimental. Use it if the input fingerprints
are in popcount order, because sorted output is a simple merge sort of
the individual sorted inputs. However, this option opens all input
files at the same time, which may exceed your resource limit on file
descriptors. The current implementation also requires a lot of disk
seeks so is slow for many files.
```

ob2fps command-line options

The following comes from `ob2fps --help`:

```
usage: ob2fps [-h]
              [--FP2 | --FP3 | --FP4 | --MACCS | --substruct | --rdmaccs | --rdmaccs/
↪1]
              [--id-tag NAME] [--in FORMAT] [-o FILENAME] [--out FORMAT]
              [--errors {strict,report,ignore}] [-R NAME=VALUE]
              [--delimiter {tab,whitespace,to-eol,space}] [--has-header]
              [--version]
              [filenames [filenames ...]]

Generate FPS or FPB fingerprints from a structure file using OpenBabel

positional arguments:
  filenames             input structure files (default is stdin)

optional arguments:
```

```
  -h, --help            show this help message and exit
  --FP2                 linear fragments up to 7 atoms
  --FP3                 SMARTS patterns specified in the file patterns.txt
  --FP4                 SMARTS patterns specified in the file
                        SMARTS_InteLigand.txt
  --MACCS               Open Babel's implementation of the MACCS 166 keys
  --substruct           ChemFP substructure fingerprints
  --rdmaccs, --rdmaccs/2
                        166 bit RDKit/MACCS fingerprints (version 2)
  --rdmaccs/1           use the version 1 definition for --rdmaccs
  --id-tag NAME         tag name containing the record id (SD files only)
  --in FORMAT           input structure format (default autodetects from the
                        filename extension)
  -o FILENAME, --output FILENAME
                        save the fingerprints to FILENAME (default=stdout)
  --out FORMAT          output structure format (default guesses from output
                        filename, or is 'fps')
  --errors {strict,report,ignore}
                        how should structure parse errors be handled?
                        (default=ignore)
  -R NAME=VALUE         specify a reader argument
  --delimiter {tab,whitespace,to-eol,space}
                        delimiter style for SMILES and InChI files. Alias for
                        '-R delimiter=VALUE'.
  --has-header          Skip the first line of a SMILES or InChI file Aliase
                        for '-R has_header=1'
  --version             show program's version number and exit

By default the Open Babel structure reader determines the file format
and compression type based on the filename extension. Unknown
filename extensions are treated as a uncompressed SMILES files.

If the data comes from stdin, or the guess based on extension name is
wrong, then use "--in FORMAT" option to change the default input format.
For examples:

   --in smi
   --in sdf.gz

The most commmon format names are :

  File Type      Valid FORMAT names
  ---------      ------------------
   SMILES        smi, can, usm  - append ".gz" for gzip'ed files
   InChI         inchi          - append ".gz" for gzip'ed files
   SDF (native)  sdf            - gzip compression is handled automatically
   SDF (chemfp)  sdf            - append ".gz"  suffix for gzip'ed files
   MOL2          mol2           - gzip compression is handled automatically
   PDB           pdb            -   "       "       "    "         "
   MacroModel    mmod           -   "       "       "    "         "

For a full list of formats, see http://openbabel.org/wiki/List_of_extensions .

Note: chemfp-2.0 removed the "ism" input format type. Use "smi" instead.

chemfp uses its own parsers to find SMILES and InChi records, which are
passed on to Open Babel for processing. These give chemfp better error
reporting and control. However, unlike the normal Open Babel parsers, they
```

```
do not automatically recognize gzip files, so the format name must include
the ".gz" suffix to read compressed formats.

By default chemfp uses Open Babel's native SDF reader. It also supports
an alternate implementation using chemfp's low-level SDF record parser.
To use chemfp's record parser, use the 'implementation' reader argument:

   -R implementation=chemfp

All format support Open Babel's 'options' OBConversion argument. This is a
compact string like 'ab"btext"', which in this case sets option 'a' to
True, and option 'b' to text "btext".

You will need to consult the Open Babel documentation and implementation
for details on the options available to each format.
```

oe2fps command-line options

The following comes from `oe2fps --help`:

```
usage: oe2fps [-h] [--path] [--circular] [--tree] [--numbits INT]
              [--minbonds INT] [--maxbonds INT] [--minradius INT]
              [--maxradius INT] [--atype ATYPE] [--btype BTYPE] [--maccs166]
              [--substruct] [--rdmaccs] [--rdmaccs/1] [--aromaticity NAME]
              [--id-tag NAME] [--in FORMAT] [-o FILENAME] [--out FORMAT]
              [--errors {strict,report,ignore}] [-R NAME=VALUE]
              [--delimiter {tab,whitespace,to-eol,space}] [--version]
              [filenames [filenames ...]]

Generate FPS or FPB fingerprints from a structure file using OEChem

positional arguments:
  filenames             input structure files (default is stdin)

optional arguments:
  -h, --help            show this help message and exit
  --aromaticity NAME    use the named aromaticity model (same as '-R
                        aromaticity=NAME')
  --id-tag NAME         tag name containing the record id (SD files only)
  --in FORMAT           input structure format (default guesses from filename)
  -o FILENAME, --output FILENAME
                        save the fingerprints to FILENAME (default=stdout)
  --out FORMAT          output structure format (default guesses from output
                        filename, or is 'fps')
  --errors {strict,report,ignore}
                        how should structure parse errors be handled?
                        (default=ignore)
  -R NAME=VALUE         specify a reader argument
  --delimiter {tab,whitespace,to-eol,space}
                        delimiter style for SMILES and InChI files. Alias for
                        '-R delimiter=VALUE'.
  --version             show program's version number and exit

path, circular, and tree fingerprints:
  --path                generate path fingerprints (default)
  --circular            generate circular fingerprints
```

```
  --tree                generate tree fingerprints
  --numbits INT         number of bits in the fingerprint (default=4096)
  --minbonds INT        minimum number of bonds in the path or tree
                        fingerprint (default=0)
  --maxbonds INT        maximum number of bonds in the path or tree
                        fingerprint (path default=5, tree default=4)
  --minradius INT       minimum radius for the circular fingerprint
                        (default=0)
  --maxradius INT       maximum radius for the circular fingerprint
                        (default=5)
  --atype ATYPE         atom type flags, described below (default=Default)
  --btype BTYPE         bond type flags, described below (default=Default)

166 bit MACCS substructure keys:
  --maccs166            generate MACCS fingerprints

881 bit ChemFP substructure keys:
  --substruct           generate ChemFP substructure fingerprints

ChemFP version of the 166 bit RDKit/MACCS keys:
  --rdmaccs, --rdmaccs/2
                        generate 166 bit RDKit/MACCS fingerprints (version 2)
  --rdmaccs/1           use the version 1 definition for --rdmaccs

ATYPE is one or more of the following, separated by the '|' character

  Arom AtmNum Chiral EqArom EqHBAcc EqHBDon EqHalo FCharge HCount HvyDeg
  Hyb InRing

The following shorthand terms and expansions are also available:
 DefaultPathAtom = AtmNum|Arom|Chiral|FCharge|HvyDeg|Hyb|EqHalo
 DefaultCircularAtom = AtmNum|Arom|Chiral|FCharge|HCount|EqHalo
 DefaultTreeAtom = AtmNum|Arom|Chiral|FCharge|HvyDeg|Hyb
and 'Default' selects the correct value for the specified fingerprint.

Examples:
  --atype Default
  --atype "Arom|AtmNum|FCharge|HCount"
  --atype Arom,AtmNum,FCharge,HCount

BTYPE is one or more of the following, separated by the '|' character

  Chiral InRing Order

The following shorthand terms and expansions are also available:
 DefaultPathBond = Order|Chiral
 DefaultCircularBond = Order
 DefaultTreeBond = Order
and 'Default' selects the correct value for the specified fingerprint.

Examples:
   --btype Default
   --btype Order|InRing

To simplify command-line use, a comma may be used instead of a '|' to
separate different fields. Example:
  --atype AtmNum,HvyDegree
```

```
OEChem guesses the input structure format based on the filename
extension and assumes SMILES for structures read from stdin.
Use "--in FORMAT" to select an alternative, where FORMAT is one of:

  File Type      Valid FORMATs (use gz if compressed)
  ---------      ------------------------------------
   SMILES        smi, can, usm, smi.gz, can.gz, usm.gz
   SDF           sdf, mol, sdf.gz, mol.gz
   SKC           skc, skc.gz
   CDK           cdk, cdk.gz
   MOL2          mol2, mol2.gz
   PDB           pdb, pdb.gz
   MacroModel    mmod, mmod.gz
   OEBinary v2   oeb, oeb.gz
   InChI         inchi, inchi.gz

Note: chemfp-2.0 removed the "ism" input format type. Use "smi" instead.

Use the '-R' reader arguments option to pass in format-specific structure
reader arguments. The details depend on the specific format. All formats
handle the following two reader arguments:

  aromaticity - one of 'openeye', 'daylight', 'tripos', 'mdl', or 'mmff'
      (this can also be set via the older '--aromaticity' command-line option)

  flavor - a '|' or ',' separated list of flavor names, or a numeric value.
       A leading '-' means to remove the given flavor. Examples include:

       o  Canon,Strict  -- the bitwise merger of the format's Canon and Strict values
       o  DEFAULT|-Kekule -- the format's DEFAULT flavor but without the Kekule bits
                     (every flavor has a DEFAULT)
       o  42  -- the specific OEChem flavor value 42

  Format    Reader arguments
  ------    ----------------
    smi,    flavor using 'Canon', 'Strict', and 'DEFAULT'
    can,    delimiter -- one of 'to-eol', 'tab', 'whitespace', or 'space'
  & usm

    sdf     the only flavor is 'DEFAULT'
    skc     the only flavor is 'DEFAULT'
    mol2    flavor using 'M2H'
   mol2h    flavor using 'M2H'
    mmod    flavor using 'FormalCrg'
    pdb     flavor using 'ALL', 'BondOrder', 'CHARGE', 'Connect', 'DATA',
                'END', 'ENDM', 'FORMALCHARGE', 'FormalCrg', 'ImplicitH',
                'RADIUS', 'Rings', 'SecStruct', and 'TER'
    xyz     flavor using 'BondOrder', 'Connect', 'FormalCrg', 'ImplicitH',
                and 'Rings'
    cdx     flavor using 'SuperAtom'
    oeb     the only flavor is 'DEFAULT'

See http://docs.eyesopen.com/toolkits/cpp/oechemtk/molreadwrite.html#flavored-input-
↪and-output
for a description of available flavors for each format.
```

rdkit2fps command-line options

The following comes from `rdkit2fps --help`:

```
usage: rdkit2fps [-h] [--fpSize FPSIZE] [--RDK] [--minPath INT]
                 [--maxPath INT] [--nBitsPerHash INT] [--useHs 0|1] [--morgan]
                 [--radius INT] [--useFeatures 0|1] [--useChirality 0|1]
                 [--useBondTypes 0|1] [--torsions] [--targetSize INT]
                 [--pairs] [--minLength INT] [--maxLength INT] [--maccs166]
                 [--avalon] [--isQuery 0_or_1] [--bitFlags INT] [--pattern]
                 [--substruct] [--rdmaccs] [--rdmaccs/1] [--id-tag NAME]
                 [--in FORMAT] [-o FILENAME] [--out FORMAT]
                 [--errors {strict,report,ignore}] [-R NAME=VALUE]
                 [--delimiter {tab,whitespace,to-eol,space}] [--has-header]
                 [--version]
                 [filenames [filenames ...]]

Generate FPS or FPB fingerprints from a structure file using RDKit

positional arguments:
  filenames             input structure files (default is stdin)

optional arguments:
  -h, --help            show this help message and exit
  --fpSize FPSIZE       number of bits in the fingerprint. Default of 2048 for
                        RDK, Morgan, topological torsion, atom pair, and
                        pattern fingerprints, and 512 for Avalon fingerprints
  --id-tag NAME         tag name containing the record id (SD files only)
  --in FORMAT           input structure format (default guesses from filename)
  -o FILENAME, --output FILENAME
                        save the fingerprints to FILENAME (default=stdout)
  --out FORMAT          output structure format (default guesses from output
                        filename, or is 'fps')
  --errors {strict,report,ignore}
                        how should structure parse errors be handled?
                        (default=ignore)
  -R NAME=VALUE         specify a reader argument
  --delimiter {tab,whitespace,to-eol,space}
                        delimiter style for SMILES and InChI files. Alias for
                        '-R delimiter=VALUE'.
  --has-header          Skip the first line of a SMILES or InChI file Aliase
                        for '-R has_header=1'
  --version             show program's version number and exit

RDKit topological fingerprints:
  --RDK                 generate RDK fingerprints (default)
  --minPath INT         minimum number of bonds to include in the subgraph
                        (default=1)
  --maxPath INT         maximum number of bonds to include in the subgraph
                        (default=7)
  --nBitsPerHash INT    number of bits to set per path (default=2)
  --useHs 0|1           include information about the number of hydrogens on
                        each atom (default=1)

RDKit Morgan fingerprints:
  --morgan              generate Morgan fingerprints
  --radius INT          radius for the Morgan algorithm (default=2)
  --useFeatures 0|1     use chemical-feature invariants (default=0)
```

```
  --useChirality 0|1    include chirality information (default=0)
  --useBondTypes 0|1    include bond type information (default=1)

RDKit Topological Torsion fingerprints:
  --torsions            generate Topological Torsion fingerprints
  --targetSize INT      number of bonds per torsion (default=4)

RDKit Atom Pair fingerprints:
  --pairs               generate Atom Pair fingerprints
  --minLength INT       minimum bond count for a pair (default=1)
  --maxLength INT       maximum bond count for a pair (default=30)

166 bit MACCS substructure keys:
  --maccs166            generate MACCS fingerprints

Avalon fingerprints:
  --avalon              generate Avalon fingerprints
  --isQuery 0_or_1      is the fingerprint for a query structure? (1 if yes, 0
                        if no) (default=0)
  --bitFlags INT        bit flags, SSSBits are 32767 and similarity bits are
                        15761407 (default=15761407)

RDKit Pattern fingerprints:
  --pattern             generate (substructure) pattern fingerprints

ChemFP's version of the 881 bit PubChem substructure keys:
  --substruct           generate ChemFP substructure fingerprints

ChemFP version of the 166 bit RDKit/MACCS keys:
  --rdmaccs, --rdmaccs/2
                        generate 166 bit RDKit/MACCS fingerprints (version 2)
  --rdmaccs/1           use the version 1 definition for --rdmaccs

This program guesses the input structure format and the compression
based on the filename extension. If the guess fails then it assumes
the input is an uncompressed SMILES file.

If the data comes from stdin, or the guess based on extension name is
wrong, then use "--in" to change the default input format. The
supported format extensions are:

  File Type      Valid FORMATs (use gz if compressed)
  ---------      ------------------------------------
   SMILES        smi, can, usm, smi.gz, can.gz, ism.gz
   SDF           sdf, sdf.gz
   InChI         inchi, inchi.gz

Note: chemfp-2.0 removed the "ism" input format type. Use "smi" instead.

Use the '-R' reader arguments option to pass in format-specific structure
reader arguments. The details depend on the specific format.

  * All of the input formats implement the 'sanitize' option. Use
    "-R sanitize=false" to disable the default sanitization.

  * The SMILES formats use the 'delimiter' option to specify the
    delimiter type. The default is 'to-eol'. The other values are
    "tab", "whitespace", and "space". Use "-R delimiter=whitespace"
```

```
    to match RDKit's native delimiter style.

  * The SDF format supports two additional reader arguments:
     * 'strictParsing'; use "-R strictParsing=false" to disable strict parsing
     * 'removeHs'; use "-R removeHs=false" to keep all of the hydrogens

  * The InChI format supports four additional reader arguments:
     * 'delimiter' works the same as it does for the SMILES formats
     * 'removeHs' works the same as it does for the SDF format
     * 'treatWarningAsError'; use "-R treatWarningAsError=true" to convert all␣
↪warnings into errors
     * 'logLevel' specifies the RDKit/InChI library log level, as an integer
```

sdf2fps command-line options

The following comes from `sdf2fps --help`:

```
usage: sdf2fps [-h] [--id-tag TAG] [--fp-tag TAG] [--in FORMAT]
               [--num-bits INT] [--errors {strict,report,ignore}]
               [-o FILENAME] [--out FORMAT] [--software TEXT] [--type TEXT]
               [--version] [--binary] [--binary-msb] [--hex] [--hex-lsb]
               [--hex-msb] [--base64] [--cactvs] [--daylight]
               [--decoder DECODER] [--pubchem]
               [filenames [filenames ...]]

Extract a fingerprint tag from an SD file and generate FPS or FPB fingerprints

positional arguments:
  filenames             input SD files (default is stdin)

optional arguments:
  -h, --help            show this help message and exit
  --id-tag TAG          get the record id from TAG instead of the first line
                        of the record
  --fp-tag TAG          get the fingerprint from tag TAG (required)
  --in FORMAT           Specify if the input SD file is uncompressed or gzip
                        compressed
  --num-bits INT        use the first INT bits of the input. Use only when the
                        last 1-7 bits of the last byte are not part of the
                        fingerprint. Unexpected errors will occur if these
                        bits are not all zero.
  --errors {strict,report,ignore}
                        how should structure parse errors be handled?
                        (default=strict)
  -o FILENAME, --output FILENAME
                        save the fingerprints to FILENAME (default=stdout)
  --out FORMAT          output structure format (default guesses from output
                        filename, or is 'fps')
  --software TEXT       use TEXT as the software description
  --type TEXT           use TEXT as the fingerprint type description
  --version             show program's version number and exit

Fingerprint decoding options:
  --binary              Encoded with the characters '0' and '1'. Bit #0 comes
                        first. Example: 00100000 encodes the value 4
  --binary-msb          Encoded with the characters '0' and '1'. Bit #0 comes
```

```
                        last. Example: 00000100 encodes the value 4
  --hex                 Hex encoded. Bit #0 is the first bit (1<<0) of the
                        first byte. Example: 01f2 encodes the value \x01\xf2 =
                        498
  --hex-lsb             Hex encoded. Bit #0 is the eigth bit (1<<7) of the
                        first byte. Example: 804f encodes the value \x01\xf2 =
                        498
  --hex-msb             Hex encoded. Bit #0 is the first bit (1<<0) of the
                        last byte. Example: f201 encodes the value \x01\xf2 =
                        498
  --base64              Base-64 encoded. Bit #0 is first bit (1<<0) of first
                        byte. Example: AfI= encodes value \x01\xf2 = 498
  --cactvs              CACTVS encoding, based on base64 and includes a
                        version and bit length
  --daylight            Daylight encoding, which is is base64 variant
  --decoder DECODER     import and use the DECODER function to decode the
                        fingerprint

shortcuts:
  --pubchem             decode CACTVS substructure keys used in PubChem. Same
                        as --software=CACTVS/unknown --type 'CACTVS-E_SCREEN/1.0␣
↪extended=2'
                        --fp-tag=PUBCHEM_CACTVS_SUBSKEYS --cactvs
```

simsearch command-line options

The following comes from `simsearch --help`:

```
usage: simsearch [-h] [-k K_NEAREST] [-t THRESHOLD] [--alpha ALPHA]
                 [--beta BETA] [--queries QUERIES] [--NxN] [--query QUERY]
                 [--hex-query HEX_QUERY] [--query-id QUERY_ID]
                 [--query-format FORMAT] [--target-format FORMAT]
                 [-o FILENAME] [-c] [-b BATCH_SIZE] [--scan] [--memory]
                 [--times] [--version]
                 target_filename

Search an FPS or FPB file for similar fingerprints

positional arguments:
  target_filename       target filename

optional arguments:
  -h, --help            show this help message and exit
  -k K_NEAREST, --k-nearest K_NEAREST
                        select the k nearest neighbors (use 'all' for all
                        neighbors)
  -t THRESHOLD, --threshold THRESHOLD
                        minimum similarity score threshold
  --alpha ALPHA         Tversky alpha parameter (default: 1.0)
  --beta BETA           Tversky beta parameter (default: the value of --alpha)
  --queries QUERIES, -q QUERIES
                        filename containing the query fingerprints
  --NxN                 use the targets as the queries, and exclude the self-
                        similarity term
  --query QUERY         query as a structure record (default format: 'smi')
  --hex-query HEX_QUERY
```

```
                        query in hex
--query-id QUERY_ID     id for the query or hex-query (default: 'Query1'
--query-format FORMAT, --in FORMAT
                        input query format (default uses the file extension,
                        else 'fps')
--target-format FORMAT
                        input target format (default uses the file extension,
                        else 'fps')
-o FILENAME, --output FILENAME
                        output filename (default is stdout)
-c, --count             report counts
-b BATCH_SIZE, --batch-size BATCH_SIZE
                        batch size
--scan                  scan the file to find matches (low memory overhead)
--memory                build and search an in-memory data structure (faster
                        for multiple queries)
--times                 report load and execution times to stderr
--version               show program's version number and exit
```

Fingerprints and fingerprint search examples

The chemfp command-line programs use a Python library called chemfp. Portions of the API are in flux and subject to change. The stable portions of the API which are open for general use are documented in *chemfp API*.

The API includes:

- low-level Tanimoto and popcount operations
- Tanimoto search algorithms based on threshold and/or k-nearest neighbors
- routines for reading and writing fingerprints
- a cross-toolkit molecule I/O API
- a cross-toolkit fingerprint type API

The following chapters give examples of how to use the API, starting with fingerprints, fingerprint I/O, and fingerprint search.

Python 2 vs. Python 3

A goal of the chemfp 3 series is to help with the transition from Python 2 to Python 3. Chemfp 3.0 was the first version of chemfp to support both major versions, that is, to support both Python 2.7 and Python 3.5 or greater. Chemfp no longer supports Python 2.5 or 2.6, though it will support Python 2.7 until 2020, which is when Python 2.7's no-cost long term support will disappear.

Previous chemfp versions, represented identifiers and fingerprints as Python (byte) strings. This was mostly okay, except when you had identifiers with non-ASCII characters.

Python 3 draws a strong distinction between text/Unicode strings and byte strings. This required some API changes in chemfp. Identifiers are now Unicode strings while fingerprints are byte strings. That one line is easy to write, but it took a few of months to implement, test, debug, and document.

The API changes are not backwards compatible. If you have code which uses the chemfp 2.x API then it may break under chemfp 3.x. Contact me if you have problems upgrading. I can help, and you pay me a support contract for a reason.

If you are writing new code which uses chemfp then you really should start using Python 3. OpenEye will stop shipping a Python 2.7 version of OEChem at the end of 2017, and RDKit will stop supporting Python 2.7 by 2020.

If you have code which works under Python 2 and you want it to work on Python 3, then there are two main options. In some cases you can re-write all the incompatible code, so the result works under Python 3 but not Python 2. However, that can be too big of a step.

Another option is to port your code to the subset of Python which works under both Python 2 and Python 3. While this is more work overall, the steps are smaller, and it's possible to develop new features while gradually doing the port.

This documentation is written with that second option in mind. The examples are shown in Python 2.7, but the same code will work under Python 3. The only differences are in the output, which I'll detail in the next section.

Unicode and byte strings

In chemfp 3.x, the record identifier is a Unicode string while the fingerprint is a byte string. Earlier versions of chemfp treated both identifiers and fingerprints as byte strings. To make things more confusing, Python 2 and Python 3 use different ways to input and denote Unicode and binary strings.

Under Python 2, normal strings are byte strings, while Unicode strings are represented with the `u""` syntax:

```
>>> "This is a byte string"      # Python 2
'This is a byte string'
>>> u"This is a Unicode string"
 u'This is a Unicode string'
```

Under Python 3, normal strings are Unicode strings, while byte strings are represented with the `b""` syntax:

```
>>> b"This is a byte string"     # Python 3
b'This is a byte string'
>>> "This is a Unicode string"
'This is a Unicode string'
```

Python 2.7 understands the `b""` notation, and Python 3 understands the `u""` notation, so the portable way to represent a Unicode identifier and binary fingerprint is to be explicit about the string type:

```
>>> id = u"España"    # Works in Python 2.7 and Python 3
>>> fp = b"\x00A!\xff"
```

While the data types are the same, the output representations are different on the two versions of Python:

```
>>> (id, fp)     # Python 2.7
(u'Espa\xf1a', '\x00A!\xff')

>>> (id, fp)     #  Python 3
('España', b'\x00A!\xff')
```

The output in these examples will be from Python 2.7. Unless otherwise stated, the equivalent output in Python 3 differs only in the prefix.

Hex representation of a binary fingerprint

In Python 2 it is easy to turn a byte string into a hex-encoded string:

```
>>> fp = b"\x00A!\xff"   # Python 2.7
>>> fp.encode("hex")
'004121ff'
```

The more direct route (and faster) is to use the binascii.hexlify function:

```
>>> import binascii    # Python 2.7
>>> binascii.hexlify(fp)
'004121ff'
```

In Python 3 it's even easier to turn a byte string into a hex-encoded string:

```
>>> fp = b"\x00A!\xff"  # Python 3
>>> fp.hex()
'004121ff'
```

but that is not portable. Nor does `fp.encode("hex")` work, because in Python 3 byte strings do not have an `encode()` method:

```
>>> fp.encode("hex")  # Python 3
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
AttributeError: 'bytes' object has no attribute 'encode'
```

If you want a byte string as output then the portable solution is to use hexlify:

```
>>> import binascii  # Python 3
>>> binascii.hexlify(fp)
b'004121ff'
```

However, on Python 2.7 I often want the hex-encoded version as a byte ("normal") string, while on Python 3 I want it as a ("normal") Unicode string, because I use hex strings for text output.

Python does not offer a portable solution, so I have added one to the *chemfp.bitops* module, named *hex_encode*

```
>>> from chemfp import bitops  # Python 2 and Python 3
>>> bitops.hex_encode(b"\x00A!\xff")
'004121ff'
```

The variant *hex_encode_as_bytes* returns a byte string, and I think is easier to remember than `binascii.hexlify`:

```
>>> bitops.hex_encode_as_bytes(b"\x00A!\xff")
b'004121ff'
```

Byte and hex fingerprints

In this section you'll learn how chemfp stores fingerprints and some of the low-level bit operations on those fingerprints.

chemfp stores fingerprints as byte strings. Here are two 8 bit fingerprints:

```
>>> fp1 = b"A"
>>> fp2 = b"B"
```

The *chemfp.bitops* module contains functions which work on byte fingerprints. Here's the *byte Tanimoto* of those two fingerprints:

```
>>> from chemfp import bitops
>>> bitops.byte_tanimoto(fp1, fp2)
0.3333333333333333
```

To understand why, you have to know that ASCII character "A" has the value 65, and "B" has the value 66. The bit representation is:

```
"A" = 01000001   and   "B" = 01000010
```

so their intersection has 1 bit and the union has 3, giving a Tanimoto of 1/3 or 0.3333333333333333 as stored in Python's 64 bit floating point value.

You can compute the Tanimoto between any two byte strings with the same length, as in:

```
>>> bitops.byte_tanimoto(b"apples&", b"oranges")
0.58333333333333334
```

You'll get a ValueError if they have different lengths:

```
>>> bitops.byte_tanimoto(b"ABC", b"A")
 Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
 ValueError: byte fingerprints must have the same length
```

The *Tversky index* is also available. The default values for alpha and beta are 1.0, which is identical to the Tanimoto:

```
>>> bitops.byte_tversky(b"apples&", b"oranges")
0.5833333333333334
>>> bitops.byte_tversky(b"apples&", b"oranges", 1.0, 1.0)
0.5833333333333334
```

Using alpha = beta = 0.5 gives the Dice index:

```
>>> bitops.byte_tversky(b"apples&", b"oranges", 0.5, 0.5)
0.7368421052631579
```

In chemfp, the alpha and beta may be between 0.0 and 100.0, inclusive. Values outside that range will raise a ValueError:

```
>>> bitops.byte_tversky(b"A", b"B", 0.2, 101)
 Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
 ValueError: beta must be between 0.0 and 100.0, inclusive
```

Most fingerprints are not as easy to read as the English ones I showed above. They tend to look more like:

```
P1@\x84K\x1aN\x00\n\x01\xa6\x10\x98\\\x10\x11
```

which is hard to read. I usually show hex-encoded fingerprints. The above fingerprint in hex is:

```
503140844b1a4e000a01a610985c1011
```

which is simpler to read. I'll use *hex_encode* as the portable way to convert a byte fingerprint to a string under Python 2 and Python 3:

```
>>> bitops.hex_encode(b"apples&")    # Portable (returns a native string)
'6170706c657326'
>>> bitops.hex_encode(b"oranges")
'6f72616e676573'
>>> bitops.hex_decode(b"416e64726577")  # (returns a byte string)
'Andrew'
```

If you do not need to support Python 2.7 then it's easier to use the Python3 specific ".hex()" and "fromhex()" methods of byte strings:

```
>>> b"apples&".hex()    # Python 3 only!
'6170706c657326'
>>> b"oranges".hex()    # Python 3 only!
'6f72616e676573'
>>> bytes.fromhex("416e64726577")    # Python 3 only!
b'Andrew'
```

Most of the byte functions in the bitops module have an equivalent hex version, like *bitops.hex_tanimoto()* which is the hex equivalent for *bitops.byte_tanimoto()*:

```
>>> bitops.hex_tanimoto("6170706c657326", "6f72616e676573")
0.5833333333333334
>>> bitops.hex_tanimoto(u"6170706c657326", u"6f72616e676573")
0.5833333333333334
>>> bitops.hex_tanimoto(b"6170706c657326", b"6f72616e676573")
0.5833333333333334
```

These functions accept both byte strings and Unicode strings.

Even though hex-encoded fingerprints are easier to read than raw bytes, it can still be hard to figure out that which bit is set in the hex fingerprint "00001000" (which is the byte fingerprint "\x00\x00\x10\x00"). For what it's worth, bit number 20 is set, where bit 0 is the first bit.

You can get the list of "on" bits with the *bitops.byte_to_bitlist()* function:

```
>>> bitops.byte_to_bitlist(b"P1@\x84K\x1aN\x00\n\x01\xa6\x10\x98\\\x10\x11")
[4, 6, 8, 12, 13, 22, 26, 31, 32, 33, 35, 38, 41, 43, 44, 49, 50,
51, 54, 65, 67, 72, 81, 82, 85, 87, 92, 99, 100, 103, 106, 107,
108, 110, 116, 120, 124]
```

That's a lot of overhead if you only want to tell if, say, bit 41 is set. For that case use *bitops.byte_contains_bit()*:

```
>>> bitops.byte_contains_bit(b"P1@\x84K\x1aN\x00\n\x01", 41)
True
>>> bitops.byte_contains_bit(b"P1@\x84K\x1aN\x00\n\x01", 42)
False
```

The *bitops.byte_from_bitlist()* function creates a fingerprint given a list of 'on' bits. By default it generates a 1024 bit fingerprint, which is a bit too long for this documentation. I'll use 64 bits instead:

```
>>> bitops.byte_from_bitlist([0], 64)
'\x01\x00\x00\x00\x00\x00\x00\x00'
```

The bit positions folded based on the modulo of the fingerprint size, so bit 65 is mapped to bit 1, as in the following:

```
>>> bitops.byte_from_bitlist([0, 65], 64)
'\x03\x00\x00\x00\x00\x00\x00\x00'
```

```
>>> bitops.byte_to_bitlist(bitops.byte_from_bitlist([0, 65], 64))
[0, 1]
```

The bitops module includes other low-level functions which work on byte fingerprints, as well as corresponding functions which work on hex fingerprints. (Hex-encoded fingerprints are decidedly second-class citizens in chemfp, but they are citizens.) The byte-based functions are:

- *byte_contains* - test if the first fingerprint is contained in the second
- *byte_contains_bit* - test if a specified fingerprint bit is on
- *byte_difference* - return a fingerprint which is the difference (xor) of two fingerprints
- *byte_from_bitlist* - create a fingerprint given 'on' bit positions
- *byte_intersect* - return a fingerprint which is the intersection of two fingerprints
- *byte_intersect_popcount* - intersection popcount between two fingerprints
- *byte_popcount* - fingerprint popcount
- *byte_tanimoto* - Tanimoto similarity between two fingerprints
- *byte_tversky* - Tversky index between two fingerprints
- *byte_to_bitlist* - get a list of the 'on' bit positions
- *byte_union* - return a fingerprint which is the union of two fingerprints
- *hex_encode* - hex encode a byte string, returns the native string type
- *hex_encode_as_bytes* - hex encode a byte string, returns a byte string

The hex-based functions are:

- *hex_contains* - test if the first hex fingerprint is contained in the second
- *hex_contains_bit* - test if a specified hex fingerprint bit is on
- *hex_difference* - return a fingerprint which is the difference (xor) of two hex fingerprints
- *hex_from_bitlist* - create a fingerprint given 'on' bit positions in a hex fingerprint
- *hex_intersect* - return a fingerprint which is the intersection of two hex fingerprints
- *hex_intersect_popcount* - intersection popcount between two hex fingerprints
- *hex_isvalid* - test if the string is a hex-encoded fingerprint
- *hex_popcount* - hex fingerprint popcount
- *hex_tanimoto* - Tanimoto similarity between two hex fingerprints
- *hex_tversky* - Tversky index between two hex fingerprints
- *hex_to_bitlist* - get a list of the 'on' bit positions in a hex fingerprint
- *hex_union* - return a fingerprint which is the union of two hex fingerprints
- *hex_decode* - convert a hex-encoded string into a byte string

There are two functions which compare a byte fingerprint to a hex fingerprint. These are somewhat faster than the pure hex version because they don't need to verify that the query fingerprint contain only hex characters:

- *byte_hex_tanimoto* - Tanimoto similarity between a byte and a hex fingerprint
- *byte_hex_tversky* - Tversky index between a byte and a hex fingerprint

Fingerprint reader and metadata

In this section you'll learn the basics of the fingerprint reader classes and fingerprint metadata.

A fingerprint record is the fingerprint plus an identifier. In chemfp, a *fingerprint reader* is an object which supports iteration through fingerprint records. There some fingerprint readers, like the *FingerprintArena* also support direct record lookup.

That's rather abstract, so let's work with a few real examples. You'll need to create a copy of the "pubchem_targets.fps" file generated in *Generate fingerprint files from PubChem SD tags* in order to follow along.

Here's how to open an FPS file:

```
>>> import chemfp
>>> reader = chemfp.open("pubchem_targets.fps")
```

Every fingerprint collection has a metadata attribute with details about the fingerprints. It comes from the header of the FPS file. You can view the metadata in Python repr format:

```
>>> reader.metadata
Metadata(num_bits=881, num_bytes=111, type=u'CACTVS-E_SCREEN/1.0 extended=2',
aromaticity=None, sources=[u'Compound_014550001_014575000.sdf.gz'],
software=u'CACTVS/unknown', date=u'2017-09-10T23:36:13')
```

In chemfp 3.x the `type`, `software`, `date` and the source filenames are Unicode strings. In earlier versions of chemfp these were byte strings.

I added a few newlines to make that easier to read, but I think it's easier still to view it in string format, which matches the format of the FPS header:

```
>>> from __future__ import print_function
>>> print(reader.metadata)
#num_bits=881
#type=CACTVS-E_SCREEN/1.0 extended=2
#software=CACTVS/unknown
#source=Compound_014550001_014575000.sdf.gz
#date=2017-09-10T23:36:13
```

(The `print` statement in Python 2 was replaced with a `print` function in Python 3. The special future statement tells Python 2 to use the new print function syntax of Python 3.)

All fingerprint collections support iteration. Each step of the iteration returns the fingerprint identifier and the fingerprint byte string. Since I know the 6th record has the id 14550010, I can write a simple loop which stops with that record:

```
>>> from chemfp import bitops
>>> for (id, fp) in reader:
...   print(id, "starts with", bitops.hex_encode(fp)[:20])
...   if id == u"14550010":
...     break
...
14550001 starts with 034e1c00020000000000
14550002 starts with 034e0c00020000000000
14550003 starts with 034e0400020000000000
14550004 starts with 03c60000000000000000
14550005 starts with 010e1c00000600000000
14550010 starts with 034e1c40000000000000
```

Fingerprint collections also support *iterating via arenas*, and several support Tanimoto search methods.

Working with a FingerprintArena

In this section you'll learn about the FingerprintArena fingerprint collection and how to iterate through subarenas in a collection.

Chemfp supports two format types. The FPS format is designed to be easy to read and write, but searching through it requires a linear scan of the disk, which can only be done once. If you want to do many queries then it's best to load the FPS data into memory as a *FingerprintArena*.

Use *chemfp.load_fingerprints()* to load fingerprints into an arena:

```
>>> from __future__ import print_function
>>> import chemfp
>>> arena = chemfp.load_fingerprints("pubchem_targets.fps")
>>> print(arena.metadata)
#num_bits=881
#type=CACTVS-E_SCREEN/1.0 extended=2
#software=CACTVS/unknown
#source=Compound_014550001_014575000.sdf.gz
#date=2017-09-10T23:36:13
```

The fingerprints can come from an FPS file, as in this example, or from an FPB file. The FPB format is much more complex internally, but can be loaded directly and quickly into a FingerprintArena, also with the same function:

```
>>> arena = chemfp.load_fingerprints("pubchem_targets.fpb")
```

An arena implements the fingerprint collection API, so you can do things like iterate over an arena and get the id/fingerprint pairs:

```
>>> from chemfp import bitops
>>> for id, fp in arena:
...     print(id, "with popcount", bitops.byte_popcount(fp))
...     if id == u"14574551":
...         break
...
14550474 with popcount 2
14574228 with popcount 2
14574262 with popcount 2
14574264 with popcount 2
14574265 with popcount 2
14574267 with popcount 2
14574635 with popcount 2
14550409 with popcount 4
14574653 with popcount 4
14550416 with popcount 6
14574831 with popcount 6
14574551 with popcount 7
```

If you look closely you'll notice that the fingerprint record order has changed from the previous section, and that the population counts are suspiciously non-decreasing. By default *load_fingerprints()* on an FPS file reorders the fingerprints into a data structure which is faster to search, though you can disable that with the *reorder* parameter if you want the fingerprints to be the same as the input order.

The *FingerprintArena* has new capabilities. You can ask it how many fingerprints it contains, get the list of identifiers, and look up a fingerprint record given an index:

```
>>> len(arena)
5167
>>> list(arena.ids[:5])
```

```
[u'14550474', u'14574228', u'14574262', u'14574264', u'14574265']
>>> id, fp = arena[6]
>>> id
u'14574635'
>>> arena[-1][0]  # the identifier of the last record in the arena
u'14564974'
>>> bitops.byte_popcount(arena[-1][1])  # its fingerprint
237
```

An arena supports iterating through subarenas. This is like having a long list and being able to iterate over sublists. Here's an example of iterating over the arena to get subarenas of size 1000 (excepting the last), and print information about each subarena:

```
>>> for subarena in arena.iter_arenas(1000):
...   print(subarena.ids[0], len(subarena))
...
14550474 1000
14570352 1000
14569340 1000
14551936 1000
14550522 1000
14570110 167
>>> arena[0][0]
u'14550474'
>>> arena[1000][0]
u'14570352'
```

To help demonstrate what's going on, I showed the first id of each record along with the main arena ids for records 0 and 1000, so you can verify that they are the same.

Arenas are a core part of chemfp. Processing one fingerprint at a time is slow, so the main search routines expect to iterate over query arenas, rather than query fingerprints.

That's why the FPSReaders – and all chemfp fingerprint collections – also support the *chemfp.FingerprintReader.iter_arenas()* method. Here's an example of reading 25 records at a time from the targets file:

```
>>> queries = chemfp.open("pubchem_queries.fps")
>>> for arena in queries.iter_arenas(25):
...   print(len(arena))
...
25
25
      <deleted additional lines saying '25'>
25
25
9
```

Those add up to 384, which you can verify is the number of structures in the original source file.

If you have a *FingerprintArena* then you can also use Python's slice notation to make a subarena:

```
>>> queries = chemfp.load_fingerprints("pubchem_queries.fps")
>>> queries[10:15]
<chemfp.arena.FingerprintArena object at 0x552c10>
>>> queries[10:15].ids
[u'27599092', u'27599227', u'27599228', u'27599115', u'27599116']
>>> queries.ids[10:15]  # a different way to get the same list
```

```
[u'27599092', u'27599227', u'27599228', u'27599115', u'27599116']
```

The big restriction is that slices can only have a step size of 1. Slices like `[10:20:2]` and `[::-1]` aren't supported. If you want something like that then you'll need to make a new arena instead of using a subarena slice. (Hint: pass the list of indices to the *arena's copy method*.)

In case you were wondering, yes, you can use `iter_arenas` and the the other FingerprintArena methods on a subarena:

```
>>> queries[10:15][1:3].ids
[u'27599227', u'27599228']
>>> queries.ids[11:13]
[u'27599227', u'27599228']
```

Save a fingerprint arena

In this section you'll learn how to save an arena in FPS and FPB formats.

This is probably the easiest section. If you have an arena (or any *FingerprintReader*), like:

```
>>> import chemfp
>>> queries = chemfp.load_fingerprints("pubchem_queries.fps")
```

then you can save it to an FPS file using the *FingerprintReader.save()* method and a filename ending with ".fps". (You'll also get an FPS file if you specify an unknown extension.):

```
>>> queries.save("example.fps")
```

If the filename ends with ".fps.gz" then the file will be saved as a gzip-compressed FPS file. Finally, if the name ends with ".fpb", as in:

```
>>> queries.save("example.fpb")
```

then the result will be in FPB format.

The `save` method supports a second option, *format*, should you for some odd reason want the format to be different than what's implied by the filename extension:

```
>>> queries.save("example.fpb", "fps")  # save in FPS format
```

How to use query fingerprints to search for similar target fingerprints

In this section you'll learn how to do a Tanimoto search using the previously created PubChem fingerprint files for the queries and the targets from *Generate fingerprint files from PubChem SD tags*.

It's faster to search an arena, so I'll load the target fingerprints:

```
>>> from __future__ import print_function
>>> import chemfp
>>> targets = chemfp.load_fingerprints("pubchem_targets.fps")
>>> len(targets)
5167
```

and open the queries as an FPSReader.

```
>>> queries = chemfp.open("pubchem_queries.fps")
```

I'll use *chemfp.threshold_tanimoto_search()* to find, for each query, all hits which are at least 0.7 similar to the query.

```
>>> for (query_id, hits) in chemfp.threshold_tanimoto_search(queries, targets,␣
↪threshold=0.7):
...   print(query_id, len(hits), list(hits)[:2])
...
27575190 3 [(4240, 0.7105263157894737), (4272, 0.7068062827225131)]
27575192 2 [(4231, 0.7157894736842105), (4773, 0.7114427860696517)]
27575198 4 [(4248, 0.703125), (4677, 0.7258883248730964)]
27575208 10 [(3160, 0.7108433734939759), (3850, 0.7102272727272727)]
27575221 8 [(3160, 0.7100591715976331), (3899, 0.7016574585635359)]
27575223 8 [(3160, 0.7100591715976331), (3899, 0.7016574585635359)]
27575240 2 [(4240, 0.7015706806282722), (4773, 0.715)]
      # ... many lines omitted ...
```

I'm only showing the first two hits for the sake of space. It seems rather pointless to show all 10 hits of query id 27575208.

However, there's a subtle problem here. The "list(hits)" returns a list of (index, score) tuples when the targets are an arena, and (id, score) tuples when the targets are a FPS reader. (I'll talk about that more in the next section for how that works.) It's best to always specify how you want the results. In my case I always want the identifiers and the scores so I'll use *hits.get_ids_and_scores()*, like this:

```
from __future__ import print_function
import chemfp
targets = chemfp.load_fingerprints("pubchem_targets.fps")
queries = chemfp.open("pubchem_queries.fps")
for (query_id, hits) in chemfp.threshold_tanimoto_search(queries, targets,␣
↪threshold=0.7):
    print(query_id, len(hits), hits.get_ids_and_scores()[:2])
```

which gives as output:

```
27575190 3 [(u'14566941', 0.7105263157894737), (u'14566938', 0.7068062827225131)]
27575192 2 [(u'14555203', 0.7157894736842105), (u'14555201', 0.7114427860696517)]
27575198 4 [(u'14552727', 0.703125), (u'14569555', 0.7258883248730964)]
27575208 10 [(u'14572463', 0.7108433734939759), (u'14560415', 0.7102272727272727)]
27575221 8 [(u'14572463', 0.7100591715976331), (u'14550456', 0.7016574585635359)]
27575223 8 [(u'14572463', 0.7100591715976331), (u'14550456', 0.7016574585635359)]
27575240 2 [(u'14566941', 0.7015706806282722), (u'14555201', 0.715)]
27575250 2 [(u'14555203', 0.7127659574468085), (u'14555201', 0.7085427135678392)]
27575257 15 [(u'14561245', 0.7218543046357616), (u'14551278', 0.7012987012987013)]
27575282 5 [(u'14566941', 0.7165775401069518), (u'14553070', 0.7070707070707071)]
27575284 0 []
    # ... many lines omitted ...
```

What you don't see in either case is that the implementation uses the *chemfp.FingerprintReader.iter_arenas()* interface on the queries so that it processes one subarena at a time. There's a tradeoff between a large arena, which is faster because it doesn't often go back to Python code, or a small arena, which uses less memory and is more responsive. You can change the tradeoff using the *arena_size* parameter.

If all you need is the count of the hits at or above a given threshold then use *chemfp.count_tanimoto_hits()*:

```
>>> queries = chemfp.open("pubchem_queries.fps")
>>> for (query_id, count) in chemfp.count_tanimoto_hits(queries, targets, threshold=0.
↪7):
```

```
...     print(query_id, count)
...
27575190 3
27575192 2
27575198 4
27575208 10
27575221 8
27575223 8
27575240 2
27575250 2
27575257 15
     # ... many lines omitted ...
```

Or, if you only want the k=2 nearest neighbors to each target within that same threshold of 0.7 then use *chemfp.knearest_tanimoto_search()*:

```
>>> queries = chemfp.open("pubchem_queries.fps")
>>> for (query_id, hits) in chemfp.knearest_tanimoto_search(queries, targets, k=2,␣
↪threshold=0.7):
...   print(query_id, hits.get_ids_and_scores())
...
27575190 [(u'14555201', 0.7236180904522613), (u'14566941', 0.7105263157894737)]
27575192 [(u'14555203', 0.7157894736842105), (u'14555201', 0.7114427860696517)]
27575198 [(u'14555201', 0.7286432160804021), (u'14569555', 0.7258883248730964)]
27575208 [(u'14555201', 0.7700534759358288), (u'14566941', 0.7584269662921348)]
27575221 [(u'14555201', 0.7591623036649214), (u'14566941', 0.7472527472527473)]
27575223 [(u'14555201', 0.7591623036649214), (u'14566941', 0.7472527472527473)]
27575240 [(u'14555201', 0.715), (u'14566941', 0.7015706806282722)]
27575250 [(u'14555203', 0.7127659574468085), (u'14555201', 0.7085427135678392)]
27575257 [(u'14572463', 0.7467532467532467), (u'14563588', 0.725)]
27575282 [(u'14555201', 0.765625), (u'14555198', 0.7317073170731707)]
27575284 []
         # ... many lines omitted ...
```

How to search an FPS file

In this section you'll learn how to search an FPS file directly, without loading it into a FingerprintArena. You'll need the previously created PubChem fingerprint files for the queries and the targets from *Generate fingerprint files from PubChem SD tags*.

The previous example loaded the fingerprints into a *FingerprintArena*. That's the fastest way to do multiple searches. Sometimes you only want to do one or a couple of queries. It seems rather excessive to read the entire targets file into an in-memory data structure before doing the search when you could search while processing the file.

For that case, use an FPSReader as the targets file. Here I'll get the first two records from the queries file and use it to search the targets file:

```
>>> from __future__ import print_function
>>> import chemfp
>>> query_arena = next(chemfp.open("pubchem_queries.fps").iter_arenas(2))
>>> query_arena
<chemfp.arena.FingerprintArena object at 0x11039c850>
>>> len(query_arena)
2
```

That first line is complicated. It opens the file and iterates over its fingerprint records two at a time as arenas. The next() returns the first of these arenas, so that line is a way of saying "get the first two records as an arena".

Here are the k=5 closest hits against the targets file:

```
>>> targets = chemfp.open("pubchem_targets.fps")
>>> for query_id, hits in chemfp.knearest_tanimoto_search(query_arena, targets, k=5,␣
↪threshold=0.0):
...   print("** Hits for", query_id, "**")
...   for hit in hits.get_ids_and_scores():
...     print("", hit)
...
** Hits for 27575190 **
 (u'14555201', 0.7236180904522613)
 (u'14566941', 0.7105263157894737)
 (u'14566938', 0.7068062827225131)
 (u'14555198', 0.6933962264150944)
 (u'14550456', 0.675531914893617)
** Hits for 27575192 **
 (u'14555203', 0.7157894736842105)
 (u'14555201', 0.7114427860696517)
 (u'14566941', 0.6979166666666666)
 (u'14566938', 0.694300518134715)
 (u'14560418', 0.6927083333333334)
```

To make it easier to see, here's the code in a single chunk:

```
from __future__ import print_function
import chemfp
query_arena = next(chemfp.open("pubchem_queries.fps").iter_arenas(2))
targets = chemfp.load_fingerprints("pubchem_targets.fps")
for query_id, hits in chemfp.knearest_tanimoto_search(query_arena, targets, k=5,␣
↪threshold=0.0):
    print("**Hits for", query_id, "**")
    for hit in hits.get_ids_and_scores():
        print("", hit)
```

Remember that the FPSReader reads an FPS file. Once you've done a search, the file is read, and you can't do another search. (Well, you *can*; but it will return empty results.) You'll need to reopen the file to reuse the file, or reseek the file handle to the start position and pass the handle to a new FPSReader.

Each search processes *arena_size* query fingerprints at a time. You will need to increase that value if you want to search more than that number of fingerprints with this method. The search performance tradeoff between an FPSReader search and loading the fingerprints into a FingerprintArena occurs at around 10 queries, so there should be little reason to worry about this.

How do to a Tversky search using the Dice weights

In this section you'll learn how to search a set of fingerprints using the more general Tversky parameters, without loading it into a FingerprintArena. You'll need the previously created PubChem fingerprint files for the queries and the targets from *Generate fingerprint files from PubChem SD tags*.

Chemfp-2.1 added support for Tversky searches. The Tversky index supports weights for the superstructure and substructure terms to the similarity. Some people like the Dice index, which is the Tversky index with alpha = beta = 0.5, so here are a couple of ways to search the targets based on the Dice index.

The previous two sections did a Tanimoto search by using *chemfp.knearest_tanimoto_search()*. The Tversky search uses *chemfp.knearest_tversky_search()*, which shouldn't be much of a surprise. Just like the Tanimoto search code, it can take a fingerprint arena or an FPS reader as the targets.

The first example loads all of the targets into an arena, then searches using each of the queries:

```
from __future__ import print_function
import chemfp
queries = chemfp.open("pubchem_queries.fps")
targets = chemfp.load_fingerprints("pubchem_targets.fps")
for query_id, hits in chemfp.knearest_tversky_search(queries, targets, k=5,
                                    threshold=0.0, alpha=0.5, beta=0.5):
    print("**Hits for", query_id, "**")
    for hit in hits.get_ids_and_scores():
        print("", hit)
```

The first two output records are:

```
**Hits for 27575190 **
 (u'14555201', 0.8396501457725948)
 (u'14566941', 0.8307692307692308)
 (u'14566938', 0.8282208588957055)
 (u'14555198', 0.8189415041782729)
 (u'14550456', 0.8063492063492064)
**Hits for 27575192 **
 (u'14555203', 0.8343558282208589)
 (u'14555201', 0.8313953488372093)
 (u'14566941', 0.8220858895705522)
 (u'14566938', 0.8195718654434251)
 (u'14560418', 0.8184615384615385)
```

On the other hand, the following reads the first two queries into an arena, then searches the targets as an FPS file, *without* loading all of the targets into memory at once:

```
import chemfp
queries = next(chemfp.open("pubchem_queries.fps").iter_arenas(2))
targets = chemfp.open("pubchem_targets.fps")
for query_id, hits in chemfp.knearest_tversky_search(queries, targets, k=5,
                                    threshold=0.0, alpha=0.5, beta=0.5):
    print("** Hits for", query_id, "**")
    for hit in hits.get_ids_and_scores():
        print("", hit)
```

Not surprisingly, this gives the same output as before:

```
** Hits for 27575190 **
 (u'14555201', 0.8396501457725948)
 (u'14566941', 0.8307692307692308)
 (u'14566938', 0.8282208588957055)
 (u'14555198', 0.8189415041782729)
 (u'14550456', 0.8063492063492064)
** Hits for 27575192 **
 (u'14555203', 0.8343558282208589)
 (u'14555201', 0.8313953488372093)
 (u'14566941', 0.8220858895705522)
 (u'14566938', 0.8195718654434251)
 (u'14560418', 0.8184615384615385)
```

FingerprintArena searches returning indices instead of ids

In this section you'll learn how to search a *FingerprintArena* and use hits based on integer indices rather than string ids.

The previous sections used a high-level interface to the Tanimoto and Tversky search code. Those are designed for the common case where you just want the query id and the hits, where each hit includes the target id.

Working with strings is actually rather inefficient in both speed and memory. It's usually better to work with indices if you can, and in the next section I'll show how to make a distance matrix using this interface.

The index-based search functions are in the *chemfp.search* module. They can be categorized into three groups, with Tanimoto and Tversky versions for each group:

1. Count the number of hits:

- `chemfp.search.count_tanimoto_hits_fp()` - search an arena using a single fingerprint (Tanimoto)
- `chemfp.search.count_tanimoto_hits_arena()` - search an arena using another arena (Tanimoto)
- `chemfp.search.count_tanimoto_hits_symmetric()` - search an arena using itself (Tanimoto)
- `chemfp.search.count_tversky_hits_fp()` - search an arena using a single fingerprint (Tversky)
- `chemfp.search.count_tversky_hits_arena()` - search an arena using another arena (Tversky)
- `chemfp.search.count_tversky_hits_symmetric()` - search an arena using itself (Tversky)

2. Find all hits at or above a given threshold, sorted arbitrarily:

- `chemfp.search.threshold_tanimoto_search_fp()` - search an arena using a single fingerprint (Tanimoto)
- `chemfp.search.threshold_tanimoto_search_arena()` - search an arena using another arena (Tanimoto)
- `chemfp.search.threshold_tanimoto_search_symmetric()` - search an arena using itself (Tanimoto)
- `chemfp.search.threshold_tversky_search_fp()` - search an arena using a single fingerprint (Tversky)
- `chemfp.search.threshold_tversky_search_arena()` - search an arena using another arena (Tversky)
- `chemfp.search.threshold_tversky_search_symmetric()` - search an arena using itself (Tversky)

3. Find the k-nearest hits at or above a given threshold, sorted by decreasing similarity:

- `chemfp.search.knearest_tanimoto_search_fp()` - search an arena using a single fingerprint (Tanimoto)
- `chemfp.search.knearest_tanimoto_search_arena()` - search an arena using another arena (Tanimoto)
- `chemfp.search.knearest_tanimoto_search_symmetric()` - search an arena using itself (Tanimoto)
- `chemfp.search.knearest_tversky_search_fp()` - search an arena using a single fingerprint (Tversky)
- `chemfp.search.knearest_tversky_search_arena()` - search an arena using another arena (Tversky)

- *chemfp.search.knearest_tversky_search_symmetric()* - search an arena using itself (Tversky)

The functions ending "_fp" take a query fingerprint and a target arena. The functions ending "_arena" take a query arena and a target arena. The functions ending "_symmetric" use the same arena as both the query and target.

In the following example, I'll use the first 5 fingerprints of a data set to search the entire data set. To do this, I load the data set as an arena, extract the first 5 records as a sub-arena, and do the search.

```
>>> from __future__ import print_function
>>> import chemfp
>>> from chemfp import search
>>> targets = chemfp.load_fingerprints("pubchem_queries.fps")
>>> queries = targets[:5]
>>> results = search.threshold_tanimoto_search_arena(queries, targets, threshold=0.7)
```

The *search.threshold_tanimoto_search_arena()* call finds the target fingerprints which have a similarity score of at least 0.7 compared to the query.

You can iterate over the results (which is a *SearchResults*) to get the list of hits for each of the queries. The order of the results is the same as the order of the records in the query:

```
>>> for hits in results:
...   print(len(hits), hits.get_ids_and_scores()[:3])
...
2 [(u'27581954', 0.9310344827586207), (u'27581957', 0.9310344827586207)]
2 [(u'27581954', 0.9310344827586207), (u'27581957', 0.9310344827586207)]
4 [(u'27580389', 1.0), (u'27580394', 0.8823529411764706), (u'27581637', 0.75)]
2 [(u'27584917', 1.0), (u'27585106', 0.8991596638655462)]
2 [(u'27584917', 0.8991596638655462), (u'27585106', 1.0)]
```

The results object don't store the query id. Instead, you have to know that the results are in the same order as the input as the query arena, so you can match the query arena's `id` attribute, which contains the list of fingerprint identifiers, to each result:

```
>>> for query_id, hits in zip(queries.ids, results):
...   print("Hits for", query_id)
...   for hit in hits.get_ids_and_scores()[:3]:
...     print("", hit)
...
Hits for 27581954
 (u'27581954', 0.9310344827586207)
 (u'27581957', 0.9310344827586207)
Hits for 27581957
 (u'27581954', 0.9310344827586207)
 (u'27581957', 0.9310344827586207)
Hits for 27580389
 (u'27580389', 1.0)
 (u'27580394', 0.8823529411764706)
 (u'27581637', 0.75)
Hits for 27584917
 (u'27584917', 1.0)
 (u'27585106', 0.8991596638655462)
Hits for 27585106
 (u'27584917', 0.8991596638655462)
 (u'27585106', 1.0)
```

What I really want to show is that you can get the same data only using the offset index for the target record instead of its id. The result from a Tanimoto search with a query arena is a *SearchResults*. Iterating over the

results gives a *SearchResult* object, with methods like *SearchResult.get_indices_and_scores()*, *SearchResult.get_ids()*, and *SearchResult.get_scores()*:

```
>>> for hits in results:
...   print(len(hits), hits.get_indices_and_scores()[:3])
...
2 [(0, 0.9310344827586207), (1, 0.9310344827586207)]
2 [(0, 0.9310344827586207), (1, 0.9310344827586207)]
4 [(2, 1.0), (5, 0.8823529411764706), (26, 0.75)]
2 [(3, 1.0), (4, 0.8991596638655462)]
2 [(3, 0.8991596638655462), (4, 1.0)]
>>>
>>> targets.ids[0]
u'27581954'
>>> targets.ids[1]
u'27581957'
>>> targets.ids[26]
u'27581637'
```

I did a few id lookups given the target dataset to show you that the index corresponds to the identifiers from the previous code.

These examples iterated over each individual *SearchResult* to fetch the ids and scores, or indices and scores. Another possibility is to ask the *SearchResults* collection to iterate directly over the list of fields you want. *SearchResults.iter_indices_and_scores()*, for example, iterates through the `get_indices_and_score` of each SearchResult.

```
>>> for row in results.iter_indices_and_scores():
...   print(len(row), row[:3])
...
2 [(0, 0.9310344827586207), (1, 0.9310344827586207)]
2 [(0, 0.9310344827586207), (1, 0.9310344827586207)]
4 [(2, 1.0), (5, 0.8823529411764706), (26, 0.75)]
2 [(3, 1.0), (4, 0.8991596638655462)]
2 [(3, 0.8991596638655462), (4, 1.0)]
```

This was added to get a bit more performance out of chemfp and because the API is sometimes cleaner one way and sometimes cleaner the other. Yes, I know that the Zen of Python recommends that "there should be one– and preferably only one –obvious way to do it." Oh well.

Computing a distance matrix for clustering

In this section you'll learn how to compute a distance matrix using the chemfp API. The next section shows an alternative way to get the similarity matrix.

chemfp does not do clustering. There's a huge number of tools which already do that. A goal of chemfp in the future is to provide some core components which clustering algorithms can use.

That's in the future, because I know little about how people want to cluster with chemfp. Right now you can use the following to build a distance matrix and pass that to one of those tools. (I'll use a distance matrix of 1 - the similarity matrix.)

Since we're using the same fingerprint arena for both queries and targets, we know the distance matrix will be symmetric along the diagonal, and the diagonal terms will be 1.0. The *chemfp.search.threshold_tanimoto_search_symmetric()* functions can take advantage of the symmetry for a factor of two performance gain. There's also a way to limit it to just the upper triangle, which cuts the memory use in half.

Most of those tools use NumPy, which is a popular third-party package for numerical computing. You will need to have it installed for the following to work.

```
import numpy  # NumPy must be installed
from chemfp import search

# Compute distance[i][j] = 1-Tanimoto(fp[i], fp[j])

def distance_matrix(arena):
    n = len(arena)

    # Start off a similarity matrix with 1.0s along the diagonal
    similarities = numpy.identity(n, "d")

    ## Compute the full similarity matrix.
    # The implementation computes the upper-triangle then copies
    # the upper-triangle into lower-triangle. It does not include
    # terms for the diagonal.
    results = search.threshold_tanimoto_search_symmetric(arena, threshold=0.0)

    # Copy the results into the NumPy array.
    for row_index, row in enumerate(results.iter_indices_and_scores()):
        for target_index, target_score in row:
            similarities[row_index, target_index] = target_score

    # Return the distance matrix using the similarity matrix
    return 1.0 - similarities
```

With the distance matrix in hand, it's easy to cluster. The SciPy package contains many clustering algorithms, as well as an adapter to generate a matplotlib graph. I'll use it to compute a single linkage clustering:

```
from __future__ import print_function
import chemfp
from scipy.cluster.hierarchy import linkage, dendrogram

 # ... insert the 'distance_matrix' function definition here ...

dataset = chemfp.load_fingerprints("pubchem_queries.fps")
distances  = distance_matrix(dataset)

linkage_matrix = linkage(distances, "single")
dendrogram(linkage_matrix,
           orientation="right",
           labels = dataset.ids)

import pylab
pylab.show()
```

Convert SearchResults to a SciPy csr matrix

In this section you'll learn how to convert a SearchResults object into a SciPy compressed sparse row matrix.

In the previous section you learned how to use the chemfp API to create a NumPy similarity matrix, and convert that into a distance matrix. The result is a dense matrix, and the amount of memory goes as the square of the number of structures.

If you have a reasonably high similarity threshold, like 0.7, then most of the similarity scores will be zero. Internally

the *SearchResults* object only stores the non-zero values for each row, along with an index to specify the column. This is a common way to compress sparse data.

SciPy has its own compressed sparse row ("csr") matrix data type, which can be used as input to many of the scikit-learn clustering algorithms.

If you want to use those algorithms, call the *SearchResults.to_csr()* method to convert the SearchResults scores (and only the scores) into a csr matrix. The rows will be in the same order as the SearchResult (and the original queries), and the columns will be in the same order as the target arena, including its ids.

I don't know enough about scikit-learn to give a useful example. (If you do, let me know!) Instead, I'll start by doing an NxM search of two sets of fingerprints:

```
from __future__ import print_function
import chemfp
from chemfp import search

queries = chemfp.load_fingerprints("pubchem_queries.fps")
targets = chemfp.load_fingerprints("pubchem_targets.fps")
results = search.threshold_tanimoto_search_arena(queries, targets, threshold = 0.8)
```

The SearchResults attribute *shape* describes the number of rows and columns:

```
>>> results.shape
(294, 5585)
>>> len(queries)
294
>>> len(targets)
5585
>>> results[6].get_indices_and_scores()
[(3304, 0.8235294117647058), (3404, 0.8115942028985508)]
```

I'll turn it into a SciPy csr:

```
>>> csr = results.to_csr()
>>> csr
<294x5585 sparse matrix of type '<type 'numpy.float64'>'
    with 87 stored elements in Compressed Sparse Row format>
>>> csr.shape
(294, 5585)
```

and look at the same row to show it has the same indices and scores:

```
>>> csr[6]
<1x5585 sparse matrix of type '<type 'numpy.float64'>'
    with 2 stored elements in Compressed Sparse Row format>
>>> csr[6].indices
array([3304, 3404], dtype=int32)
>>> csr[6].data
array([ 0.82352941,  0.8115942 ])
```

Taylor-Butina clustering

For the last clustering example, here's my (non-validated) variation of the Butina algorithm from JCICS 1999, 39, 747-750. See also http://www.redbrick.dcu.ie/~noel/R_clustering.html . You might know it as Leader clustering.

First, for each fingerprint find all other fingerprints with a threshold of 0.8:

```
from __future__ import print_function
import chemfp
from chemfp import search

arena = chemfp.load_fingerprints("pubchem_targets.fps")
results = search.threshold_tanimoto_search_symmetric(arena, threshold = 0.8)
```

Sort the results so that fingerprints with more hits come first. This is more likely to be a cluster centroid. Break ties arbitrarily by the fingerprint id; since fingerprints are ordered by the number of bits this likely makes larger structures appear first:

```
# Reorder so the centroid with the most hits comes first.
# (That's why I do a reverse search.)
# Ignore the arbitrariness of breaking ties by fingerprint index
results = sorted( (  (len(indices), i, indices)
                          for (i, indices) in enumerate(results.iter_indices())  ),
                  reverse=True)
```

Apply the leader algorithm to determine the cluster centroids and the singletons:

```
# Determine the true/false singletons and the clusters
true_singletons = []
false_singletons = []
clusters = []

seen = set()
for (size, fp_idx, members) in results:
    if fp_idx in seen:
        # Can't use a centroid which is already assigned
        continue
    seen.add(fp_idx)

    # Figure out which ones haven't yet been assigned
    unassigned = set(members) - seen

    if not unassigned:
        false_singletons.append(fp_idx)
        continue

    # this is a new cluster
    clusters.append( (fp_idx, unassigned) )
    seen.update(unassigned)
```

Once done, report the results:

```
print(len(true_singletons), "true singletons")
print("=>", " ".join(sorted(arena.ids[idx] for idx in true_singletons)))
print()

print(len(false_singletons), "false singletons")
print("=>", " ".join(sorted(arena.ids[idx] for idx in false_singletons)))
print()

# Sort so the cluster with the most compounds comes first,
# then by alphabetically smallest id
def cluster_sort_key(cluster):
    centroid_idx, members = cluster
    return -len(members), arena.ids[centroid_idx]
```

```
clusters.sort(key=cluster_sort_key)

print(len(clusters), "clusters")
for centroid_idx, members in clusters:
    print(arena.ids[centroid_idx], "has", len(members), "other members")
    print("=>", " ".join(arena.ids[idx] for idx in members))
```

The algorithm is quick for this small data set.

Out of curiosity, I tried this on 100,000 compounds selected arbitrarily from PubChem. It took 35 seconds on my desktop (a 3.2 GHZ Intel Core i3) with a threshold of 0.8. In the Butina paper, it took 24 hours to do the same, although that was with a 1024 bit fingerprint instead of 881. It's hard to judge the absolute speed differences of a MIPS R4000 from 1998 to a desktop from 2011, but it's less than the factor of about 2000 you see here.

More relevent is the comparison between these numbers for the 1.1 release compared to the original numbers for the 1.0 release. On my old laptop, may it rest it peace, it took 7 minutes to compute the same benchmark. Where did the roughly 16-fold peformance boost come from? Money. After 1.0 was released, Roche funded various optimizations, including taking advantage of the symmetery (2x) and using hardware POPCNT if available (4x). Roche and another company helped fund the OpenMP support, and when my desktop reran this benchmark it used 4 cores instead of 1.

The wary among you might notice that 2*4*4 = 32x faster, while I said the overall code was only 16x faster. Where's the factor of 2x slowdown? It's in the Python code! The *chemfp.search.threshold_tanimoto_search_symmetric()* step took only 13 seconds. The remaining 22 seconds was in the leader code written in Python. To make the analysis more complicated, improvements to the chemfp API sped up the clustering step by about 40%.

With chemfp 1.0 version, the clustering performance overhead was minor compared to the full similarity search, so I didn't keep track of it. With chemfp 1.1, those roles have reversed!

Configuring OpenMP threads

In this section you'll learn about chemfp and OpenMP threads, including how to set the number of threads to use.

OpenMP is an API for shared memory multiprocessing programming. Chemfp uses it to parallelize the similarity search algorithms. Support for OpenMP is a compile-time option for chemfp, and can be disabled with *--without-openmp* in setup.py. Versions 4.2 of gcc (released in 2007) and later support it, as do other compilers, though chemfp has only been tested with gcc.

Chemfp uses one thread per query fingerprint. This means that single fingerprint queries are not parallelized. There is no performance gain even if four cores are available.

(A note about nomenclature: a CPU can have one core, or it can have several cores. A single processor computer has one CPU while a multiprocessor computer has several CPUs. I think some cores can even run multiple threads. So it's possible to have many more hardware threads than CPUs.)

Chemfp uses multiple threads when there are many queries, which occurs when using a query arena against a target arena. These search methods include the high-level API in the top-level chemfp module (like 'knearest_tanimoto_search'), and the arena search function in chemfp.search.

By default, OpenMP and therefore chemfp will use four threads:

```
>>> import chemfp
>>> chemfp.get_num_threads()
4
```

You can change this through the standard OpenMP environment variable OMP_NUM_THREADS in the shell:

```
% env OMP_NUM_THREADS=2 python
Python 2.6.7 (r267:88850, Oct  9 2013, 03:47:03)
[GCC 4.2.1 (Apple Inc. build 5666) (dot 3)] on darwin
Type "help", "copyright", "credits" or "license" for more information.
>>> import chemfp
>>> chemfp.get_num_threads()
2
```

or you can specify the number of threads directly using set_num_threads():

```
>>> chemfp.set_num_threads(3)
>>> chemfp.get_num_threads()
3
```

If you specify 0 or 1 thread then chemfp will not use OpenMP at all and stick with a single-threaded implementation. (You probably want to disable OpenMP in multi-threaded programs like web servers. See the next section for details.)

Throwing more threads at a task doesn't always make it faster. My desktop has one CPU with two cores, so it's pointless to have more than two OpenMP threads running, as you can see from some timings:

```
threshold_tanimoto_search_symmetric (threshold=0.8) (desktop)
#threads  time (in s)
   1       22.6
   2       13.1
   3       12.3
   4       12.9
   5       12.6
```

On the other hand, my laptop has 1 CPU with four cores, and while my desktop beats my laptop with single threaded peformance, once I have three cores going, my laptop is faster:

```
threshold_tanimoto_search_symmetric (threshold=0.8) (laptop)
#threads  time (in s)
   1       27.4
   2       14.6
   3       10.3
   4        8.2
   5        9.0
```

How many cores/hardware threads are available? That's a really good question. chemfp implements `chemp.get_max_threads()`, but that doesn't seem to do what I want. So don't use it, and I'll figure out a real solution in a future release.

OpenMP and multi-threaded applications

In this section you'll learn some of the problems of mixing OpenMP and multi-threaded code.

Do not use OpenMP and multiple threads on a Mac. It will crash. This includes Django, which is a multi-threaded web server. In multi-threaded code on a Mac you must either tell chemfp to be single-threaded, using:

```
chemfp.set_num_threads(1)
```

or figure out some way to put the chemfp search code into its own process space, which is a much harder solution.

Other OSes will let you mix POSIX and OpenMP threads, but life gets confusing. Might your web server handle three search requests at the same time? If so, should all of those get four OpenMP threads, so that 12 threads are running in total? Can your hardware handle that many threads?

It may be better to have chemfp not use OpenMP threads when under a multi-threaded system, or have some way to limit the number of chemfp search tasks running at the same time. Figuring out the right solution will depend on your hardware and requirements.

Fingerprint Substructure Screening (experimental)

In this section you'll learn how to find target fingerprints which contain the query fingerprint bit patterns as a subset. Bear in mind that this is an experimental API.

Substructure search often uses a screening step to remove obvious mismatches before doing the subgraph isomorphism. One way is to generate a binary fingerprint such that if a query molecule is a substructure of a target molecule then the corresponding query fingerprint is completely contained in the target fingerprint, that is, the target fingerprint must have 'on' bits for all of the query fingerprints which have 'on' bits.

I'll start by loading a fingerprint arena with four fingerprints, where the identifiers are Unicode strings and the fingerprint are byte strings of length 1, with the binary form shown to the right:

```
>>> from __future__ import print_function
>>> import chemfp
>>> from chemfp import bitops
>>> arena = chemfp.load_fingerprints([
...     (u"A1", b"\x44"),    # 0b01000100
...     (u"B2", b"\x6c"),    # 0b01101100
...     (u"C3", b"\x95"),    # 0b10010101
...     (u"D4", b"\xea"),    # 0b11101010
...   ], chemfp.Metadata(num_bits=8))
>>> for id, fp in arena:
...   print(bitops.hex_encode(fp), id)
...
44 A1
6c B2
95 C3
ea D4
```

I could use *bitops.byte_contains()* to search for fingerprints in a loop, in this case with a query fingerprint which requires that the 7th bit be set (they must fit the pattern `0b*1******`):

```
>>> query_fingerprint = b"\x40"    # 0b01000000
>>> bitops.hex_encode(query_fingerprint)
'40'
>>> for id, target_fingerprint in arena:
...   if bitops.byte_contains(query_fingerprint, target_fingerprint):
...     print(id)
...
A1
B2
D4
```

This is slow because it uses Python to do almost all of the work. Instead, use *contains_fp()* from the *chemfp.search* module, which is faster because it's all implemented in C:

```
>>> from chemfp import search
>>> result = search.contains_fp(query_fingerprint, arena)
>>> result
<chemfp.search.SearchResult object at 0x10195e090>
>>> print(result.get_ids())
[u'A1', u'B2', u'D4']
```

This is the same *SearchResult* instance that the similarity search code returns, though the scores are all 0.0:

```
>>> result.get_ids_and_scores()
[(u'A1', 0.0), (u'B2', 0.0), (u'D4', 0.0)]
```

This API is experimental and likely to change. Please provide feedback. While I don't think the current call parameters will change, I might have it return the Tanimoto score (or Hamming distance?) instead of 0.0. Or I might have a way to compute new scores given a SearchResult.

I also plan to support start/end parameters, to search only a subset of the arena.

There's also a *search.contains_arena()* function which takes a query arena instead of only a query fingerprint as the query, and returns a *SearchResults*:

```
>>> results = search.contains_arena(arena, arena)
>>> results
<chemfp.search.SearchResults object at 0x10195c2b8>
>>> for result in results:
...   print(result.get_ids_and_scores())
...
[(u'A1', 0.0), (u'B2', 0.0)]
[(u'B2', 0.0)]
[(u'C3', 0.0)]
[(u'D4', 0.0)]
```

I don't think the NxN version of the "contains" search is all that useful, so there's no function for that case.

The implementation doesn't yet support OpenMP, `contains_arena()` is only little faster than multiple calls to `contains_fp()`.

Substructure screening with RDKit

In this section you'll learn how to use RDKit's pattern fingerprint (in development) for substructure screening.

Of the three toolkits that chemfp supports, only RDKit has fingerprint tuned for substructure search, though it's marked as 'experimental' and subject to change. This is the "pattern" fingerprint.

I'll use it to make a screen for one of the PubChem files. Normally you would start with something like:

```
% rdkit2fps --pattern Compound_014550001_014575000.sdf.gz -o pubchem_screen.fpb
```

but that only gives me the identifiers and fingerprints. I want to show some of the struture as well, so I'll do a bit of a cheat - I'll have an augmented identifier which is the PubChem id, followed by a space, followed by the SMILES string.

I can do this because chemfp supports almost anything as the "identifier", except newline, tab, and the NUL character, and because I don't need to support id lookup.

However, I have to write Python code to generate the augmented identifiers:

```
import chemfp

fptype = chemfp.get_fingerprint_type("RDKit-Pattern fpSize=1024")
T = fptype.toolkit

with chemfp.open_fingerprint_writer("pubchem_screen.fpb", fptype.get_metadata()) as␣
↪writer:
  for id, mol in T.read_ids_and_molecules("Compound_014550001_014575000.sdf.gz"):
    smiles = T.create_string(mol, "canstring")  # use the non-isomeric SMILES string
```

```
    fp = fptype.compute_fingerprint(mol)
    # Create an "identifier" of the form:
    #    PubChem id + " " + canonical SMILES string
    writer.write_fingerprint(id + " " + smiles, fp)
```

Now that I have the screen, I'll write some code to actually do the screen. I'll make this be an interactive prompt, which asks for the query SMILES string (or "quit" or "exit" to quit), parses the SMILES to a molecule, generates the fingerprint, does the screen, and displays the first 10 results:

```
from __future__ import print_function
import itertools
import chemfp
import chemfp.search

fptype = chemfp.get_fingerprint_type("RDKit-Pattern fpSize=1024")
T = fptype.toolkit

screen = chemfp.load_fingerprints("pubchem_screen.fpb")
print("Loaded", len(screen), "screen fingerprints")

while 1:
  # Ask for the query SMILES string
  query = raw_input("Query? ")
  if query in ("quit", "exit"):
    break

  # See if it's a valid SMILES
  mol = T.parse_molecule(query, "smistring", errors="ignore")
  if mol is None:
    print("Could not parse query")
    continue

  # Compute the fingerprint and do the substructure screeening
  fp = fptype.compute_fingerprint(mol)
  result = chemfp.search.contains_fp(fp, screen)

  # Print the results, up to 10.
  n = len(result)
  if n > 10:
    print(len(result), "matches. First 10 displayed")
    n = 10
  else:
    print(len(result), "matches.")

  for augmented_id in itertools.islice(result.iter_ids(), 0, n):
    id, smiles = augmented_id.split()
    print(id, "=>", smiles)
  print()
```

(In case you haven't seen it before, the "itertools.islice()" gives me an easy way to get up to the first N items from an iterator.)

I'll try out the above code:

```
Loaded 5208 screen fingerprints
Query? c1ccccc1
3476 matches. First 10 displayed
14571805 => SCCOCc1ccccc1
```

```
14574154 => Cl[Ti]Cl.c1ccccc1.c1ccccc1
14571795 => SCCCSCc1ccccc1
14568980 => C[C](O)c1ccccc1
14568981 => CC([O-])c1ccccc1
14571571 => ICCC(I)c1ccccc1
14573102 => [N-]=[N+]=Nc1cccc(N=[N+]=[N-])c1
14567762 => CC(O)CCC#Cc1ccccc1
14568647 => ClC(Cl)CCOc1ccccc1
14567111 => CCCC(=Cc1ccccc1)CO

Query? c1ccccc1O
1274 matches. First 10 displayed
14568647 => ClC(Cl)CCOc1ccccc1
14557991 => C=CC=COCc1ccc(OC)cc1
14572069 => C=CCNC(=S)Oc1ccccc1
14550766 => NCCNCC(O)COc1ccccc1
14572073 => C=CCNC(=O)Oc1ccccc1
14572952 => Cc1ccc(OCO)c(C)c1
14550768 => NCCCNCC(O)COc1ccccc1
14574927 => CC(N)COc1cccc(Cl)c1
14570157 => CC=CCOc1ccccc1CCC
14567814 => CNc1ccc(OC)cc1C

Query? c1ccccc1I
6 matches.
14573520 => Nc1cc(N)c(I)cc1I
14566147 => S=c1[nH]c2ccc(I)cc2[nH]1
14571184 => O=[N+]([O-])c1cccc([N+](=O)[O-])c1I
14567222 => COn1ccc2cccc(I)c21
14566148 => O=C(O)CSc1nc2ccc(I)cc2[nH]1
14572760 => Ic1ccc(N=c2snc3sc4ccccc4n23)nc1

Query? CC(C1=C(C(=C(C(=C1F)F)F)F)F)Br
1 matches.
14550341 => CC(Br)c1c(F)c(F)c(F)c(F)c1F

Query? quit
```

Looks reasonable.

It's not hard to add full substructure matching, but it requires toolkit-specific code. Chemfp doesn't try to abstract that detail, and I'm not sure it should be part of chemfp. Instead, I'll write some RDKit-specific code. Chemfp uses native toolkit molecules, so there's actually only a single line of RDKit code.

I'll also completely rewrite the code so it takes the query string on the command-line, reports all of the screening results, identifies the true positives, and then does a brute-force verification that the screen results are correct. Oh, and report statistics:

```
# This program is called 'search.py'
from __future__ import print_function
import sys
import chemfp
import chemfp.search
from chemfp import rdkit_toolkit as T # Will only work with RDKit
import time

fptype = chemfp.get_fingerprint_type("RDKit-Pattern fpSize=1024")
```

```
screen = chemfp.load_fingerprints("pubchem_screen.fpb")
if len(sys.argv) != 2:
  raise SystemExit("Usage: %s <smiles>" % (sys.argv[0],))

query_smiles = sys.argv[1]

start_time = time.time()
try:
  query_mol = T.parse_molecule(query_smiles, "smistring")
except ValueError as err:
  raise SystemExit(str(err))

# Compute the fingerprint and do the substructure screeening
fp = fptype.compute_fingerprint(query_mol)
result = chemfp.search.contains_fp(fp, screen)
search_time = time.time()

num_matches = 0

for augmented_id in result.get_ids():
  id, smiles = augmented_id.split()
  target_mol = T.parse_molecule(smiles, "smistring")
  if target_mol.HasSubstructMatch(query_mol):   # RDKit specific!
    print(id, "matches", smiles)
    num_matches += 1
  else:
    print(id, "       ", smiles)
report_time = time.time()

# Report the results
print()
print("= Screen search =")
print("num targets:", len(screen))
print("screen size:", len(result))
print("num matches:", num_matches)
print("screenout: %.1f%%" % (100.0 * (len(screen)-len(result)) / len(screen),))
if len(result) == 0:
  precision = 100.0
else:
  precision = (100.0*num_matches) / len(result)
print("precision: %.1f%%" % (precision,))
print("screen time: %.2f" % (search_time - start_time,))
print("atom-by-atom-search and report time: %.2f" % (report_time - search_time,))
print("total time: %.2f" % (report_time - start_time,))

# Reduce the computations without any screening
num_actual = 0
actual_start_time = time.time()
for augmented_id in screen.ids:
  id, smiles = augmented_id.split()
  target_mol = T.parse_molecule(smiles, "smistring")
  if target_mol.HasSubstructMatch(query_mol):   # RDKit specific!
    num_actual += 1
actual_end_time = time.time()

print()
print("= Brute force search =")
print("num matches:", num_actual)
```

```
print("time to test all molecules: %.2f" % (actual_end_time - actual_start_time,))
print("screening speedup: %.1f" % ((actual_end_time - actual_start_time) / (report_
↪time - start_time),))
```

Here's the output with 'c1ccccc1O' on the command-line:

```
% python search.py c1ccccc1O
14568647 matches ClC(Cl)CCOc1ccccc1
14557991 matches C=CC=COCc1ccc(OC)cc1
14572069 matches C=CCNC(=S)Oc1ccccc1
14550766 matches NCCNCC(O)COc1ccccc1
14572073 matches C=CCNC(=O)Oc1ccccc1
14572952 matches Cc1ccc(OCO)c(C)c1
14550768 matches NCCCNCC(O)COc1ccccc1
   ...
14565454 matches␣
↪CCOCCC(=O)Oc1c2cccc(OC3OC(C)C4OC(c5ccccc5)OC4C3OC3OC(C)C(O)C(OC)C3O)c2c2oc(=O)c3c(C)ccc4oc(=O)c1c2c
14565455 matches␣
↪CCOCCC(=O)Oc1c2cccc(OC3OC(C)C4OC(c5ccccc5)OC4C3OC3OC(C)C(O)C(OC)C3O)c2c2oc(=O)c3c(C)ccc4oc(=O)c1c2c
14558058 matches CCCCCCCCCCCCCCCc1c2[nH]c(nc3nc(nc4nc(nc5[nH]c1c1cc(OCC(C)(C)C)ccc51)-
↪c1cc(OCC(C)(C)C)ccc1-4)-c1cc(OCC(C)(C)C)ccc1-3)c1cc(OCC(C)(C)C)ccc21

= Screen search =
num targets: 5208
screen size: 1274
num matches: 1248
screenout: 75.5%
precision: 98.0%
screen time: 0.00
atom-by-atom-search and report time: 0.71
total time: 0.71

= Brute force search =
num matches: 1248
time to test all molecules: 2.01
screening speedup: 2.8
```

It's a relief to see that the versions with and without the screen give the same number of matches!

Next, 'c1ccccc1I' (that's iodobenzene):

```
% python search.py 'c1ccccc1I'
14573520 matches Nc1cc(N)c(I)cc1I
14566147 matches S=c1[nH]c2ccc(I)cc2[nH]1
14571184 matches O=[N+]([O-])c1cccc([N+](=O)[O-])c1I
14567222 matches COn1ccc2cccc(I)c21
14566148 matches O=C(O)CSc1nc2ccc(I)cc2[nH]1
14572760         Ic1ccc(N=c2snc3sc4ccccc4n23)nc1

= Screen search =
num targets: 5208
screen size: 6
num matches: 5
screenout: 99.9%
precision: 83.3%
screen time: 0.00
atom-by-atom-search and report time: 0.00
total time: 0.00
```

```
= Brute force search =
num matches: 5
time to test all molecules: 2.03
screening speedup: 506.3
```

Now for some bad news. Try '[Pu]'. This doesn't screen out many structures yet has no matched. I'll report the search statistics:

```
.. code-block:: none
```

= Screen search = num targets: 5208 screen size: 5160 num matches: 0 screenout: 0.9% precision: 0.0% screen time: 0.00 atom-by-atom-search and report time: 2.30 total time: 2.31

= Brute force search = num matches: 0 time to test all molecules: 1.85 screening speedup: 0.8

That's horrible! It's slower! What happened is that '[Pu]' generates a fingerprint with only two bits set:

```
% echo '[Pu] plutonium' | rdkit2fps --pattern --fpSize 1024
#FPS1
#num_bits=1024
#type=RDKit-Pattern/4 fpSize=1024
#software=RDKit/2017.09.1.dev1 chemfp/3.1
#date=2017-09-14T23:11:48
0000000020000000000000000000000000000000000000000000000000000000000000000000000000000000
0000800000000000000000000000000000000000000000000000000000000000000000000000000000000000
0000000000000000000000000000000000000000000000000000000000000000000000000000000000000000
0000000000000000      plutonium
```

You know, that's really hard to see. I'll use a bit of perl to replace the zeros with ".":

```
% echo '[Pu] plutonium' | python ../rdkit2fps --pattern --fpSize 1024 | perl -pe 's/0/
↪./g'
#FPS1
#num_bits=1.24
#type=RDKit-Pattern/4 fpSize=1.24
#software=RDKit/2.17..9.1.dev1 chemfp/3.1
#date=2.17-.9-14T23:29:44
........2.......................................................................
....8...........................................................................
................................................................................
................      plutonium
```

Ha! And it converted zeros in the header lines to ".". I'll just omit the header lines in the following.

Unfortunately, so many other structures also set those two bits, like the following two:

```
% echo 'CNn1cccn1 test1' | python ../rdkit2fps --pattern --fpSize 1024 | perl -pe 's/
↪0/./g'
...2...83.......1..8....1.1.1..2.82...4.222.........1..34.....28..c........1.2..
...19.8.........6..7......2.1...1...22.4...4.....18a1...2.184............44a....
......38.28..244...8..3.1...1...2...2............4..2....a.....8.44....c8..24.1.
c8181..4.a18..4.      test1

% echo 'Cc1cccccc1=S test2' | python ../rdkit2fps --pattern --fpSize 1024 | perl -pe
↪'s/0/./g'
...2....2.42...15.......1....4...82...4.222....1..241..24.....6..28........1.2..
....b..1..25...86..7......331.8.....22.4..15.....188..2.2.8...28...1......c6....
```

```
.........6..8244.......411......3..422....8.4...........124......4...1.8...2..2.
881.4....a.8..4.      test2
```

Reading structure fingerprints using a toolkit

In this section you'll learn how to use a chemistry toolkit to compute fingerprints from a given structure file.

What happens if you're given a structure file and you want to find the two nearest matches in an FPS file? You'll have to generate the fingerprints for the structures in the structure file, then do the comparison.

For this section you'll need to have a chemistry toolkit. I'll use the "chebi_maccs.fps" file generated in *Using a toolkit to process the ChEBI dataset* as the targets, and the PubChem file Compound_027575001_027600000.sdf.gz as the source of query structures:

```
>>> from __future__ import print_function
>>> import chemfp
>>> from chemfp import search
>>> targets = chemfp.load_fingerprints("chebi_maccs.fps")
>>> queries = chemfp.read_molecule_fingerprints(targets.metadata, "Compound_027575001_
↪027600000.sdf.gz")
>>> for (query_id, hits) in chemfp.knearest_tanimoto_search(queries, targets, k=2,␣
↪threshold=0.0):
...   print(query_id, "=>", end=" ")
...   for (target_id, score) in hits.get_ids_and_scores():
...     print("%s %.3f" % (target_id, score), end=" ")
...   print()
...
27575190 => CHEBI:116551 0.779 CHEBI:105622 0.771
27575192 => CHEBI:105622 0.809 CHEBI:108425 0.809
27575198 => CHEBI:109833 0.736 CHEBI:105937 0.730
27575208 => CHEBI:105622 0.783 CHEBI:108425 0.783
27575240 => CHEBI:91516 0.747 CHEBI:111326 0.737
27575250 => CHEBI:105622 0.809 CHEBI:108425 0.809
27575257 => CHEBI:105622 0.732 CHEBI:108425 0.732
     # ... many lines omitted ...
```

That's it! Pretty simple, wasn't it? I didn't even need to explicitly specify which toolkit I wanted to use because the *read_molecule_fingerprints()* got that information from the arena's *Metadata*.

The new function is *chemfp.read_molecule_fingerprints()*, which reads a structure file and generates the appropriate fingerprints for each one. The first parameter of this is the metadata used to configure the reader. In my case it's:

```
>>> print(targets.metadata)
#num_bits=166
#type=OpenBabel-MACCS/2
#software=OpenBabel/2.4.1 chemfp/3.1
#source=ChEBI_lite.sdf.gz
#date=2017-09-16T00:15:13
```

The metadata's "type" told chemfp which toolkit to use to read molecules, and how to generate fingerprints from those molecules. (Note: the "aromaticity" value is no longer in use. The original version of OEGraphSim used the user-defined aromaticity model, which meant that the same structure could give different results depending on the file format used. OEGraphSim v2 now always re-perceives using OpenEye's aromaticity model.)

You can pass in your own metadata as the first parameter to `read_molecule_fingerprints`, and as a shortcut, if you pass in a string then it will be used as the fingerprint type.

For examples, if you have OpenBabel installed then you can do:

```
>>> from chemfp import bitops
>>> reader = chemfp.read_molecule_fingerprints("OpenBabel-MACCS", "Compound_027575001_
↪027600000.sdf.gz")
>>> for i, (id, fp) in enumerate(reader):
 ...   print(id, bitops.hex_encode(fp))
 ...   if i == 3:
 ...     break
 ...
27575190 000000000020449e8401c148a0f4122cfa8a7bff1f
27575192 000000000000449e8401c148e0f4122cfa8a6bff1f
27575198 000000000000449e8401914838f9122edb8a3bff1f
27575208 000000040000449e8401c148a0f4122cfa8a6bff1f
```

If you have OEChem and OEGraphSim installed and licensed then you can do:

```
>>> from chemfp import bitops
>>> reader = chemfp.read_molecule_fingerprints("OpenEye-MACCS166", "Compound_
↪027575001_027600000.sdf.gz")
>>> for i, (id, fp) in enumerate(reader):
...   print(id, bitops.hex_encode(fp))
...   if i == 3:
...     break
...
27575190 000000080020448e8401c148a0f41216fa8a7b7e1b
27575192 000000080000448e8401c148e0f41216fa8a6b7e1b
27575198 000000080000448e8401d14838f91216db8a3b7e1b
27575208 0000000c0000448e8401c148a0f41216fa8a6b7e1b
```

And if you have RDKit installed then you can do:

```
>>> from chemfp import bitops
>>> reader = chemfp.read_molecule_fingerprints("RDKit-MACCS166", "Compound_027575001_
↪027600000.sdf.gz")
>>> for i, (id, fp) in enumerate(reader):
...   print(id, bitops.hex_encode(fp))
...   if i == 3:
...     break
...
27575190 000000000020449e8401c148a0f4123cfa8a7bff1f
27575192 000000000000449e8401c148e0f4123cfa8a6bff1f
27575198 000000000000449e8401914838f9123edb8a3bff1f
27575208 000000040000449e8401c148a0f4123cfa8a6bff1f
```

Select a random fingerprint sample

In this section you'll learn how to make a new arena where the fingerprints are randomly selected from the old arena.

A *FingerprintArena* slice creates a subarena. Technically speaking, this is a "view" of the original data. The subarena doesn't actually copy its fingerprint data from the original arena. Instead, it uses the same fingerprint data, but keeps track of the start and end position of the range it needs. This is why it's not possible to slice with a step size other than +1.

This also means that memory for a large arena won't be freed until all of its subarenas are also removed.

You can see some evidence for this because a *FingerprintArena* stores the entire fingerprint data as a set of bytes named `arena`:

```
>>> import chemfp
>>> targets = chemfp.load_fingerprints("pubchem_targets.fps")
>>> subset = targets[10:20]
>>> targets.arena is subset.arena
True
```

This shows that the *targets* and *subset* share the same raw data set. At least it does to me, the person who wrote the code.

You can ask an arena or subarena to make a *copy*. This allocates new memory for the new arena and copies all of its fingerprints there.

```
>>> new_subset = subset.copy()
>>> len(new_subset) == len(subset)
True
>>> new_subset.arena is subset.arena
False
>>> subset[7][0]
'14571646'
>>> new_subset[7][0]
'14571646'
```

The `copy` method can do more than just copy the arena. You can give it a list of indices and it will only copy those fingerprints:

```
>>> three_targets = targets.copy([3112, 0, 1234])
>>> three_targets.ids
['14550474', '14570519', '14570965']
>>> [targets.ids[3112], targets.ids[0], targets.ids[1234]]
['14570965', '14550474', '14570519']
```

Are you confused about why the identifiers aren't in the same order? That's because when you specify indicies, the copy automatically reorders them by popcount and stores the popcount information. This requires a bit extra overhead to sort, but makes future searches faster. Use `reorder=False` to leave the order unchanged

```
>>> my_ordering = targets.copy([3112, 0, 1234], reorder=False)
>>> my_ordering.ids
['14570965', '14550474', '14570519']
```

Let's get back to the main goal of getting a random subset of the data. I want to select *m* records at random, without replacement, to make a new data set. You can see this just means making a list with *m* different index values. Python's built-in random.sample function makes this easy:

```
>>> import random
>>> random.sample("abcdefgh", 3)
['b', 'h', 'f']
>>> random.sample("abcdefgh", 2)
['d', 'a']
>>> random.sample([5, 6, 7, 8, 9], 2)
[7, 9]
>>> help(random.sample)
sample(self, population, k) method of random.Random instance
   Chooses k unique random elements from a population sequence.
   ...
   To choose a sample in a range of integers, use xrange as an argument.
   This is especially fast and space efficient for sampling from a
   large population:   sample(xrange(10000000), 60)
```

The last line of the help points out what do next!:

```
>>> random.sample(xrange(len(targets)), 5)
[610, 2850, 705, 1402, 2635]
>>> random.sample(xrange(len(targets)), 5)
[1683, 2320, 1385, 2705, 1850]
```

Putting it all together, and here's how to get a new arena containing 100 randomly selected fingerprints, without replacement, from the *targets* arena:

```
>>> sample_indices = random.sample(xrange(len(targets)), 100)
>>> sample = targets.copy(indices=sample_indices)
>>> len(sample)
100
```

Don't reorder an arena by popcount

In this section you'll learn about why you might want to store your fingerprints in specific order, rather than being ordered by population count.

The previous section showed how to make an arena where the fingerprints are in a user-specified order:

```
>>> import chemfp
>>> targets = chemfp.load_fingerprints("pubchem_targets.fps")
>>> [targets.ids[i] for i in [3112, 0, 1234]]
[u'14556313', u'14550474', u'14566849']
>>> targets.copy([3112, 0, 1234], reorder=False).ids
[u'14556313', u'14550474', u'14566849']
>>> targets.copy([3112, 0, 1234], reorder=True).ids
[u'14550474', u'14566849', u'14556313']
```

If the `reorder` option is not specified, the fingerprints in the new arena will be in popcount order. Similarity search is faster when the arena is in popcount order because it lets chemfp make an index of the different regions, based on popcount, and use that for sublinear search.

Why would someone want search to be slower?

Sometimes data organization is more important. For one client I developed a SEA implementation, where I compared a set of query fingerprints to about 50 other sets of target fingerprint sets. The largest set had only few thousand fingerprints, so the overall search was fast without a popcount index.

I could have stored each target data set as its own file, but that would have resulted in about 50 data files to manage, in addition to the original fingerprint file and the configuration file containing the information about which identifiers are in which set.

Instead, I stored all of the target data sets in a single FPB file, where the fingerprints for the first set came first, then the fingerprints for the second set, and so on. I also made a range file to store the set name and the start/end range of that set in the FPB file. This reduced 50 files down to two, which was much easier to manage.

It's a bit fiddly to go through the details of how this works, because it requires set membership information which is a bit complicated to extract and which won't be used for the rest of this documentation. Instead of walking though an example here, I'll refer you to my essay ChEMBL target sets association network.

You can use the subranges directly as an arena slice, like `arena[54:91]` as the target. This will work, but as I said earlier, the search time will be slower because the sublinear algorithm requires a popcount index.

If you need that search performance then during load time make a copy of the slice, as in `arena[54:91].copy(reorder=True)`, and use that as the target.

A few paragraphs ago I wrote that "I stored all of the target data sets in a single FPB file." When you load an FPB format, the fingerprint order will be exactly as given in the file. However, if you load fingerprints from an FPS file, the fingerprints are by default reordered. For example, given this data set:

```
% cat unordered_example.fps
#FPS1
0001  Record1
ffee  Record2
00f0  Record3
```

I'll load it into chemfp and show that by default the records are in the order 1, 3, 2:

```
>>> import chemfp
>>> chemfp.load_fingerprints("unordered_example.fps").ids
chemfp.load_fingerprints("unordered_example.fps").ids
```

On the other hand, if I ask it to not reorder then the records are in the input order, which is 1, 2, 3:

```
>>> chemfp.load_fingerprints("unordered_example.fps", reorder=False).ids
[u'Record1', u'Record2', u'Record3']
```

In short, if you want to preserve the fingerprint order as given in the input file then use the `reorder=False` argument in *`chemfp.load_fingerprints()`*.

Look up a fingerprint with a given id

In this section you'll learn how to get a fingerprint record with a given id. You will need the "pubchem_targets.fps" file generated in *Generate fingerprint files from PubChem SD tags* in order to do this yourself.

All fingerprint records have an identifier and a fingerprint. Identifiers should be unique. (Duplicates are allowed, and if they exist then the lookup code described in this section will arbitrarily decide which record to return. Once made, the choice will not change.)

Let's find the fingerprint for the record in "pubchem_targets.fps" which has the identifier "14564126". One solution is to iterate over all of the records in a file, using the FPS reader:

```
>>> import chemfp
>>> for id, fp in chemfp.open("pubchem_targets.fps"):
...   if id == "14564126":
...     break
... else:
...   raise KeyError("%r not found" % (id,))
...
>>> fp[:5]
'\x07\x1e\x1c\x00\x00'
```

(Under Python 3 that last line will show a `b''` string because fingerprints are byte strings.)

I used the somewhat obscure `else` clause to the `for` loop. If the `for` finishes without breaking, which would happen if the identifier weren't present, then it will raise an exception saying that it couldn't find the given identifier.

If the fingerprint records are already in a *`FingerprintArena`* then there's a better solution. Use the *`FingerprintArena.get_fingerprint_by_id()`* method to get the fingerprint byte string, or None if the identifier doesn't exist:

```
>>> arena = chemfp.load_fingerprints("pubchem_targets.fps")
>>> fp = arena.get_fingerprint_by_id("14564126")
>>> fp[:5]
```

```
'\x07\x1e\x1c\x00\x00'
>>> missing_fp = arena.get_fingerprint_by_id("does-not-exist")
>>> missing_fp
>>> missing_fp is None
True
```

Internally this does about what you think it would. It uses the arena's `id` list to make a lookup table mapping identifier to index, and caches the table for later use. Given the index, it's very easy to get the fingerprint.

In fact, you can get the index and do the record lookup yourself:

```
>>> arena.get_index_by_id("14564126")
2824
>>> arena[2820]
(u'14564126', '\x07\x1e\x1c\x00\x00 ...')
```

Sorting search results

In this section you'll learn how to sort the search results.

The k-nearest searches return the hits sorted from highest score to lowest, and break ties arbitrarily. This is usually what you want, and the extra cost to sort is small (k*log(k)) compared to the time needed to maintain the internal heap (N*log(k)).

By comparison, the threshold searches return the hits in arbitrary order. Sorting takes up to N*log(N) time, which is extra work for those cases where you don't want sorted data. If you actually want it sorted, then call *`SearchResult.reorder()`* method to sort the hits in-place:

```
>>> import chemfp
>>> arena = chemfp.load_fingerprints("pubchem_queries.fps")
>>> query_fp = arena.get_fingerprint_by_id("27599116")
>>> from chemfp import search
>>> result = search.threshold_tanimoto_search_fp(query_fp, arena, threshold=0.90)
>>> len(result)
9
>>> result.get_ids_and_scores()
[('27599061', 0.953125), ('27599092', 0.9615384615384616),
('27599227', 0.9615384615384616), ('27599228',
0.9615384615384616), ('27599115', 1.0), ('27599116', 1.0),
('27599118', 1.0), ('27599120', 1.0), ('27599082',
0.9253731343283582)]
>>>
>>> result.reorder("decreasing-score")
>>> result.get_ids_and_scores()
[('27599115', 1.0), ('27599116', 1.0), ('27599118', 1.0),
('27599120', 1.0), ('27599092', 0.9615384615384616), ('27599227',
0.9615384615384616), ('27599228', 0.9615384615384616),
('27599061', 0.953125), ('27599082', 0.9253731343283582)]
>>>
>>> result.reorder("increasing-score")
>>> result.get_ids_and_scores()
[('27599082', 0.9253731343283582), ('27599061', 0.953125),
('27599092', 0.9615384615384616), ('27599227',
0.9615384615384616), ('27599228', 0.9615384615384616),
('27599115', 1.0), ('27599116', 1.0), ('27599118', 1.0),
('27599120', 1.0)]
```

There are currently six different sort methods, all specified by a name string. These are

- increasing-score - sort by increasing score
- decreasing-score - sort by decreasing score
- increasing-index - sort by increasing target index
- decreasing-index - sort by decreasing target index
- reverse - reverse the current ordering
- move-closesot-first - move the hit with the highest score to the first position

The first two should be obvious from the examples. If you find something useful for the next two then let me know. The "reverse" method reverses the current ordering, and is most useful if you want to reverse the sorted results from a k-nearest search.

The "move-closest-first" option exists to improve the leader algorithm stage used by the Taylor-Butina algorithm. The newly seen compound is either in the same cluster as its nearest neighbor or it is the new centroid. I felt it best to implement this as a special reorder term, rather than one of the other possible options.

If you have suggestions for alternate orderings which might help improve your clustering performance, let me know.

If you want to reorder all of the search results then you could use the *SearchResult.reorder()* method on each result, but it's easier to use *SearchResults.reorder_all()* and change everything in a single call. It takes the same ordering names as `reorder`:

```
>>> from __future__ import print_function
>>> similarity_matrix = search.threshold_tanimoto_search_symmetric(
...                         arena, threshold=0.8)
>>> for query_id, row in zip(arena.ids, similarity_matrix):
...   print(query_id, "->", row.get_ids_and_scores()[:3])
...
27581954 -> [('27581957', 0.9310344827586207)]
27581957 -> [('27581954', 0.9310344827586207)]
27580389 -> [('27580394', 0.8823529411764706)]
27584917 -> [('27585106', 0.8991596638655462)]
27585106 -> [('27584917', 0.8991596638655462)]
27580394 -> [('27580389', 0.8823529411764706)]
27599061 -> [('27599092', 0.9453125), ('27599227', 0.9453125), ('27599228', 0.
↪9453125)]
27593061 -> []
27575880 -> [('27575997', 0.8194444444444444)]
27583796 -> []
27599092 -> [('27599227', 0.9689922480620154), ('27599228', 0.9689922480620154), (
↪'27599115', 0.9615384615384616)]
       ... lines deleted ....
>>>
>>> similarity_matrix.reorder_all("increasing-score")
>>> for query_id, row in zip(arena.ids, similarity_matrix):
...   print(query_id, "->", row.get_ids_and_scores()[:3])
...
27581954 -> [('27581957', 0.9310344827586207)]
27581957 -> [('27581954', 0.9310344827586207)]
27580389 -> [('27580394', 0.8823529411764706)]
27584917 -> [('27585106', 0.8991596638655462)]
27585106 -> [('27584917', 0.8991596638655462)]
27580394 -> [('27580389', 0.8823529411764706)]
27599061 -> [('27598934', 0.8), ('27599095', 0.8108108108108109), ('27598670', 0.
↪8137931034482758)]
27593061 -> []
```

```
27575880 -> [('27575997', 0.8194444444444444)]
27583796 -> []
27599092 -> [('27598959', 0.8108108108108109), ('27598934', 0.8211920529801324), (
↪'27598670', 0.8231292517006803)]
       ... lines deleted ....
```

These are almost identical because most of the searches have only zero or one hits. The only differences are in the lines for ids "27599061" and "27599092", both of which have 19 hits. For display purposes, I used `[:3]` to display only the first three matches. In the first block the results are in arbitrary order, while in the second the elements are sorted so the smallest score is first.

Working with raw scores and counts in a range

In this section you'll learn how to get the hit counts and raw scores for an interval.

The length of a *SearchResult* is the number of hits it contains:

```
>>> import chemfp
>>> from chemfp import search
>>> arena = chemfp.load_fingerprints("pubchem_targets.fps")
>>> fp = arena.get_fingerprint_by_id("14564126")
>>> result = search.threshold_tanimoto_search_fp(fp, arena, threshold=0.2)
>>> len(result)
4682
```

This gives you the number of hits at or above a threshold of 0.2, which you can also get by doing *chemfp.search.count_tanimoto_hits_fp()*:

```
>>> search.count_tanimoto_hits_fp(fp, arena, threshold=0.2)
4682
```

The advantage to the first version is the result also stores the hits. You can query the hit to get the number of hits which are within a specified interval. Here are the counts of the number of hits at or above 0.5, 0.80, and 0.95:

```
>>> result.count(0.5)
1218
>>> result.count(0.8)
9
>>> result.count(0.95)
2
```

The first parameter, *min_score*, specifies the minimum threshold. If not specified it's -infinity. The second, *max_score*, specifies the maximum, and is +infinity if not specified. Here's how to get the number of hits with a score of at most 0.95 and 0.5:

```
>>> result.count(max_score=0.95)
4680
>>> result.count(max_score=0.5)
3489
```

If you double-check the math, and add the number above 0.5 (1218) and the number below 0.5 (3489) you'll get 4707, even through there are only 4682 records. The extra 25 is because by default the count interval uses a closed range. There are 25 hits with a score of exactly 0.5:

```
>>> result.count(0.5, 0.5)
25
```

The third parameter, *interval*, specifies the end conditions. The default is "[]" which means that both ends are closed. The interval "()" means that both ends are open, and "[)" and "(]" are the two half-open/half-closed ranges. To get the number of hits below 0.5 and the number of hits at or above 0.5 then you might use:

```
>>> result.count(None, 0.5, "[)")
3722
>>> result.count(0.5, None, "[]")
1364
>>> 3464+1218
4682
```

This total matches the expected count. (A min or max of *None* means -infinity and +infinity, respectively.)

Cumulative search result counts and scores

In this section you'll learn some more advanced ways to work with SearchResults and SearchResult instances.

I wanted to title this section "Going to SEA", but decided to use a more descriptive name. "SEA" refers to the "Similarity Ensemble Approach" (SEA) work of Keiser, Roth, Armbruster, Ernsberger, and Irwin. The paper is available online from http://sea.bkslab.org/ , though I won't actually implement it here. Why do I mention it? Because these chemfp methods were added specifically to make it easier to support a SEA implementation for one of the chemfp customers.

Suppose you have two sets of structures. How well do they compare to each other? I can think of various ways to do it. One is to look at a comparison profile. Find all NxM comparisons between the two sets. How many of the hits have a threshold of 0.2? How many at 0.5? 0.95?

If there are "many", then the two sets are likely more similar than not. If the answer is "few", then they are likely rather distinct.

I'll be more specific. I want to know if the coenzyme A-like structures in ChEBI are more similar to the penicillin-like structures than one would expect by comparing two randomly chosen subsets. To quantify "similar", I'll use Tanimoto similarity of the "chebi_maccs.fps" fingerprints, which are the 166 MACCS key-like fingerprints from RDMACCS for the ChEBI data set. See *Using a toolkit to process the ChEBI dataset* for details about why I use the `--id-tag` options:

```
# Use one of the following to create chebi_maccs.fps
oe2fps --id-tag "ChEBI ID" --rdmaccs ChEBI_lite.sdf.gz -o chebi_maccs.fps
ob2fps --id-tag "ChEBI ID" --rdmaccs ChEBI_lite.sdf.gz -o chebi_maccs.fps
rdkit --id-tag "ChEBI ID" --rdmaccs ChEBI_lite.sdf.gz -o chebi_maccs.fps
```

I used oe2fps to create RDMACCS-OpenEye fingerprints.

The CHEBI id for coenzyme A is CHEBI:15346 and for penicillin is CHEBI:17334. I'll define the "coenzyme A-like" structures as the 256 structures where the fingerprint is at least 0.95 similar to coenzyme A, and "penicillin-like" as the 24 structures at least 0.85 similar to penicillin. This gives 6144 total comparisons.

You know enough to do this, but there's a nice optimization I haven't told you about. You can get the total count of all of the threshold hits using the *chemfp.search.SearchResults.count_all()* method instead of looping over each SearchResult and calling *chemfp.search.SearchResult.count()*:

```
from __future__ import print_function
import chemfp
from chemfp import search

def get_neighbors_as_arena(arena, id, threshold):
    fp = arena.get_fingerprint_by_id(id)
    neighbor_results =  search.threshold_tanimoto_search_fp(fp, chebi,␣
↪threshold=threshold)
```

```
    neighbor_arena = arena.copy(neighbor_results.get_indices())
    return neighbor_arena

chebi = chemfp.load_fingerprints("chebi_maccs.fps")

# Find the 256 neighbors of coenzyme A
coA_arena = get_neighbors_as_arena(chebi, "CHEBI:15346", threshold=0.95)
print(len(coA_arena), "coenzyme A-like structures")

# Find the 24 neighbors of penicillin
penicillin_arena = get_neighbors_as_arena(chebi, "CHEBI:17334", threshold=0.85)
print(len(penicillin_arena), "penicillin-like structures")

# I'll compute a profile at different thresholds
thresholds = [0.25, 0.5, 0.6, 0.7, 0.8, 0.9, 0.95]

# Compare the two sets. (For this case the speed difference between a threshold
# of 0.25 and 0.0 is not noticible, but having it makes me feel better.)
coA_against_penicillin_result = search.threshold_tanimoto_search_arena(
    coA_arena, penicillin_arena, threshold=min(thresholds))

# Show a similarity profile
print("Counts  coA/penicillin")
for threshold in thresholds:
    print(" %.2f      %5d" % (threshold,
                              coA_against_penicillin_result.count_all(min_
↪score=threshold)))
```

This gives a not very useful output:

```
261 coenzyme A-like structures
8 penicillin-like structures
Counts  coA/penicillin
 0.30       2088
 0.35       2088
 0.40       2087
 0.45       1113
 0.50          0
 0.60          0
 0.70          0
 0.80          0
 0.90          0
```

It's not useful because it's not possible to make any decisions from this. Are the numbers high or low? It should be low, because these are two quite different structure classes, but there's nothing to compare it against.

I need some sort of background reference. What I'll do is construct two randomly chosen sets, one with 256 fingerprints and the other with 24, and generate the same similarity profile with them. That isn't quite fair, since randomly chosen sets will most likely be diverse. Instead, I'll pick one fingerprint at random, then get its 256 or 24, respectively, nearest neighbors as the set members (place the following code at the end of the file with the previous code):

```
# Get background statistics for random similarity groups of the same size
import random

# Find a fingerprint at random, get its k neighbors, return them as a new arena
def get_random_fp_and_its_k_neighbors(arena, k):
    fp = arena[random.randrange(len(arena))][1]
    similar_search = search.knearest_tanimoto_search_fp(fp, arena, k)
```

```
    return arena.copy(similar_search.get_indices())
```

I'll construct 1000 pairs of sets this way, accumulate the threshold profile, and compare the CoA/penicillin profile to it:

```
# Initialize the threshold counts to 0
total_background_counts = dict.fromkeys(thresholds, 0)

REPEAT = 1000
for i in range(REPEAT):
    # Select background sets of the same size and accumulate the threshold count␣
↪totals
    set1 = get_random_fp_and_its_k_neighbors(chebi, len(coA_arena))
    set2 = get_random_fp_and_its_k_neighbors(chebi, len(penicillin_arena))
    background_search = search.threshold_tanimoto_search_arena(set1, set2,␣
↪threshold=min(thresholds))
    for threshold in thresholds:
        total_background_counts[threshold] += background_search.count_all(min_
↪score=threshold)

print("Counts  coA/penicillin  background")
for threshold in thresholds:
    print(" %.2f       %5d           %5d" % (threshold,
                                             coA_against_penicillin_result.count_
↪all(min_score=threshold),
                                             total_background_counts[threshold] /␣
↪(REPEAT+0.0)))
```

Your output should now have something like this at the end:

```
Counts  coA/penicillin  background
 0.30       2088            882
 0.35       2088            698
 0.40       2087            550
 0.45       1113            413
 0.50          0            322
 0.60          0            156
 0.70          0             58
 0.80          0             20
 0.90          0              5
```

This is a bit hard to interpret. Clearly the coenzyme A and penicillin sets are not closely similar, but for low Tanimoto scores the similarity is higher than expected. That difficulty is okay for now because I mostly wanted to show an example of how to use the chemfp API. If you want to dive deeper into this sort of analysis then read a three-part series I wrote at http://www.dalkescientific.com/writings/diary/archive/2017/03/20/fingerprint_set_similarity.html on using chemfp to build a target set association network using ChEMBL.

The SEA paper actually wants you to use the raw score, which is the sum of the hit scores in a given range, and not just the number of hits. No problem! Use *SearchResult.cumulative_score()* for the cumulative scores for an individual result, or *SearchResults.cumulative_score_all()* for the cumulative scores across all of the results. The two functions compute *almost* identical values for the whole data set:

```
>>> sum(row.cumulative_score(min_score=0.5, max_score=0.9)
...             for row in coA_against_penicillin_result)
582.129474678352
>>> coA_against_penicillin_result.cumulative_score_all(min_score=0.5, max_score=0.9)
582.1294746783535
```

The cumulative methods, like the count method you learned about in the previous section, also take the *interval* parameter for when you don't want the default of "[]".

You may wonder why these two values aren't exactly the same. They differ because floating point addition is not associative. The first computes the sum for each result, then the sum of sums. The second computes the sum by adding each score to the cumulative sum.

I get a different result if I sum up the values in reverse order:

```
>>> sum(list(row.cumulative_score(min_score=0.5, max_score=0.9)
...                for row in coA_against_penicillin_result)[::-1])
582.1294746783539
```

Which is the "right" score? The `cumulative_score_all()` method at least matches the one you might write if you computed the sum directly:

```
>>> total_score = 0.0
>>> for row_scores in coA_against_penicillin_result.iter_scores():
...   for score in row_scores:
...     if 0.5 <= score <= 0.9:
...       total_score += score
...
>>> total_score
582.1294746783535
```

Writing fingerprints with a fingerprint writer

In this section you'll learn how to create a fingerprint file using the chemfp fingerprint writer API.

You probably don't need this section. In most cases you can save the contents of an FPS reader or fingerprint arena by using the *`FingerprintReader.save()`* method, as in the following examples:

```
chemfp.open("pubchem_targets.fps").save("example.fps")
chemfp.open("pubchem_targets.fps").save("example.fpb")
chemfp.open("pubchem_targets.fpb").save("example.fps.gz")
```

The structure-based fingerprint readers also implement the `save` method so you could simply write:

```
import chemfp
reader = chemfp.read_molecule_fingerprints("RDKit-MACCS166", "Compound_014550001_
↪014575000.sdf.gz")
reader.save("example.fps")  # or "example.fpb"
```

However, if you generate the fingerprints yourself, or want more fine-grained control over the writer parameters then read on!

(If you don't have RDKit installed then use "OpenBabel-MACCS" for Open Babel's MACCS fingerprints, and "OpenEye-MACCS166" for OpenEye's.)

Here's an example of the fingerprint writer API. I open the writer, ask it to write a fingerprint id and the fingerprint, and then close it.

```
>>> import chemfp
>>> writer = chemfp.open_fingerprint_writer("example.fps")
>>> writer.write_fingerprint("ABC123", b"\0\0\0\0\0\3\2\1")
>>> writer.close()
```

I'll ask Python to read the file and print the contents:

```
>>> from __future__ import print_function
>>> print(open("example.fps").read())
#FPS1
0000000000030201      ABC123
```

Of course you don't need to use chemfp to write this file. It's simple enough that you could get the same result in fewer lines of normal Python code. The advantage starts to be useful when you want to include metadata.

```
>>> metadata = chemfp.Metadata(num_bits=64, type="Example-FP/0")
>>> writer = chemfp.open_fingerprint_writer("example.fps", metadata)
>>> writer.write_fingerprint("ABC123", b"\0\0\0\0\0\3\2\1")
>>> writer.close()
>>>
>>> print(open("example.fps").read())
#FPS1
#num_bits=64
#type=Example-FP/0
0000000000030201      ABC123
```

Even then, native Python code is probably easier to use if you know what the header lines will be, because it's a bit of a nuisance to create the *chemfp.Metadata* yourself.

On the other hand, if you have a chemfp fingerprint type you can just ask it for the correct metadata instance:

```
>>> fptype = chemfp.get_fingerprint_type("RDKit-MACCS166")
>>> metadata = fptype.get_metadata()
>>> metadata
Metadata(num_bits=166, num_bytes=21, type='RDKit-MACCS166/2',
aromaticity=None, sources=[], software='RDKit/2017.09.1.dev1 chemfp/3.1',
date='2017-09-16T00:01:50')
```

Putting the two together, and switching to a 21 byte fingerprint instead of an 8 byte fingerprint, gives:

```
>>> from __future__ import print_function
>>> import chemfp
>>> fptype = chemfp.get_fingerprint_type("RDKit-MACCS166")
>>> writer = chemfp.open_fingerprint_writer("example.fps", fptype.get_metadata())
>>> writer.write_fingerprint("ABC123", b
↪"\0\1\2\3\4\5\6\7\x08\x09\x0A\x0B\x0C\x0D\x0E\x0F\x10\x11\x12\x13\x14")
>>> writer.close()
>>>
>>> print(open("example.fps").read())
#FPS1
#num_bits=166
#type=RDKit-MACCS166/2
#software=RDKit/2017.09.1.dev1 chemfp/3.1
#date=2017-09-16T00:02:29
000102030405060708090a0b0c0d0e0f1011121314    ABC123
```

In real life that fingerprint comes from somewhere. The high-level structure-based fingerprint reader has a handy `metadata` attribute:

```
>>> filename = "Compound_014550001_014575000.sdf.gz"
>>> reader = chemfp.read_molecule_fingerprints("RDKit-MACCS166", filename)
>>> print(reader.metadata)
#num_bits=166
#type=RDKit-MACCS166/2
#software=RDKit/2017.09.1.dev1 chemfp/3.1
```

```
#source=Compound_014550001_014575000.sdf.gz
#date=2017-09-16T00:03:14
```

By the way, note that this includes the source filename, which *FingerprintType.get_metadata()* can't automatically do. (See *Merging multiple structure-based fingerprint sources* for an example of how to pass that information to get_metadata().)

A structure-based fingerprint reader is just like any other reader, so you can iterate over the (id, fingerprint) pairs:

```
>>> from chemfp import bitops
>>> reader = chemfp.read_molecule_fingerprints("RDKit-MACCS166", filename)
>>> for count, (id, fp) in enumerate(reader):
...   print(id, "=>", bitops.hex_encode(fp))
...   if count == 5:
...     break
...
14550001 => 000080000000081406000a226a010614a5fceae7d1f
14550002 => 00000000000000000000aa06801021405dc6e47d1f
14550003 => 00000000000000000000a8160000054054c4e0bd1f
14550004 => 000000000000800118204a00000800900b1708813
14550005 => 0000000004080100000000001001001400803523e
14550010 => 00000018008400001000304 4a882000e8e0e0a771f
```

You probably already see how to combine this with *FingerprintWriter.write_fingerprint()* to generate the FPS output. The key part would look like:

```
for id, fp in reader:
  writer.write_fingerprint(id, fp)
```

While that would work, there's a better way. The chemfp fingerprint writer has a *FingerprintWriter.write_fingerprints()* method which takes a list or iterator of (id, fingerprint) pairs. Here's a better way to write the code:

```
import chemfp
filename = "Compound_014550001_014575000.sdf.gz"
reader = chemfp.read_molecule_fingerprints("RDKit-MACCS166", "Compound_014550001_
↪014575000.sdf.gz")
writer = chemfp.open_fingerprint_writer("example.fps", reader.metadata)
writer.write_fingerprints(reader)
writer.close()
reader.close()
# Note: See the next section for an even better  solution
# which uses a context manager.
```

This produces output which starts:

```
#FPS1
#num_bits=166
#type=RDKit-MACCS166/2
#software=RDKit/2017.09.1.dev1 chemfp/3.1
#source=Compound_014550001_014575000.sdf.gz
#date=2017-09-16T00:04:23
000080000000081406000a226a010614a5fceae7d1f    14550001
00000000000000000000aa06801021405dc6e47d1f    14550002
00000000000000000000a8160000054054c4e0bd1f    14550003
000000000000800118204a00000800900b1708813    14550004
0000000004080100000000001001001400803523e    14550005
```

Why is `write_fingerprints` "better" than multiple calls to `write_fingerprint`? I think it more directly describes the goal of writing all of the fingerprints, rather than the mechanics of unpacking and repacking the (id, fingerprint) pairs. I had hoped that there would be performance improvement, because there's less Python function call overhead, but my timings show no differences.

However, there's a still better way, which is to use a context manager to close the files automatically, rather than calling `close()` explicitly. I'll leave that for the next section.

Fingerprint readers and writers are context managers

In this section you'll learn how the fingerprint readers and writers can be used as a context manager.

The previous section ended with the following code:

```
import chemfp
filename = "Compound_014550001_014575000.sdf.gz"
reader = chemfp.read_molecule_fingerprints("RDKit-MACCS166", filename)
writer = chemfp.open_fingerprint_writer("example.fps", reader.metadata)
writer.write_fingerprints(reader)
writer.close()
reader.close()
```

This reads a PubChem file with RDKit, generates MACCS fingerprints, and saves the results to "example.fps".

The two *FingerprintWriter.close()* lines ensure that the reader and writer files are closed. This isn't required for a simple script, because Python will close the files automatically at the end of the script, or when the garbage collector kicks in.

However, since the writer may buffer the output, you have to close the file before you or another program can read it. It's good practice to always close the file when you're done with it, as otherwise there are ways to get really confused about why you don't have a complete file.

Even with the explicit `close` calls, if there's an exception in *FingerprintWriter.write_fingerprints()* then the files will be left open. In older-style Python this was handled with a try/finally block, but that's verbose. Instead, chemfp's readers and writers implement modern Python's context manager API, to make it easier to close files automatically at just the right place. Here's what the above looks like with a context manager:

```
import chemfp
filename = "Compound_014550001_014575000.sdf.gz"
with chemfp.read_molecule_fingerprints("RDKit-MACCS166", filename) as reader:
  with chemfp.open_fingerprint_writer("example.fps", reader.metadata) as writer:
    writer.write_fingerprints(reader)
```

Isn't that nice and short? Just bear in mind that it's even more succinctly written as:

```
import chemfp
filename = "Compound_014550001_014575000.sdf.gz"
with chemfp.read_molecule_fingerprints("RDKit-MACCS166", filename) as reader:
  reader.save("example.fps")
```

Write fingerprints to stdout or a file-like object

In this section you'll learn how to write fingerprints to stdout, and how to write them to a BytesIO instance.

The previous section showed examples of passing a filename string to *chemfp.open_fingerprint_writer()*. If the `filename` argument is None then the writer will write to stdout in uncompressed FPS format:

```
>>> import chemfp
>>> writer = chemfp.open_fingerprint_writer(None,
...     chemfp.Metadata(num_bits=16, type="Experiment/1"))
#FPS1
#num_bits=16
#type=Experiment/1
>>> writer.write_fingerprint("QWERTY", b"AA")
4141    QWERTY
>>> writer.write_fingerprint("SHRDLU", b"\0\1")
0001    SHRDLU
>>> writer.close()
```

The `filename` argument may also be a file-like object, which is defined as any object which implements the method `write(s)` where `s` is a byte string. A io.BytesIO instance is one such file-like object. It gives access to the output as a byte string:

```
>>> from __future__ import print_function
>>> import chemfp
>>> from io import BytesIO
>>> f = BytesIO()
>>> writer = chemfp.open_fingerprint_writer(f, chemfp.Metadata(num_bits=16, type=
↪"Experiment/1"))
>>> print(f.getvalue())
#FPS1
#num_bits=16
#type=Experiment/1

>>> writer.write_fingerprint("ETAOIN", b"00")
>>> writer.close()
>>> print(f.getvalue())
#FPS1
#num_bits=16
#type=Experiment/1
3030    ETAOIN
```

(Note: Under Python 3 the two `print(f.getvalue())` lines will display the byte string as:

```
b'#FPS1\n#num_bits=16\n#type=Experiment/1\n'
b'#FPS1\n#num_bits=16\n#type=Experiment/1\n3030\tETAOIN\n'
```

You can see that closing the fingerprint writer does not close the underlying file-like object. (If it did then you couldn't get access to the string content, which gets deleted when the StringIO is closed.)

You can also write an FPB file to a file-like object, if it supports `seek()` and `tell()` and binary writes. This means that you cannot write an FPB format to stdout, but you can write it to a BytesIO instance.

```
>>> import chemfp
>>> from io import BytesIO
>>> f = BytesIO()
>>> writer = chemfp.open_fingerprint_writer(f, format="fpb")
>>> writer.write_fingerprint("ID123", b"\x01\xfe")
>>> writer.close()
>>> len(f.getvalue())
2269
```

Writing fingerprints to an FPB file

In this section you'll learn how to write an FPB file.

The FPS file is a text format which was designed to be easy to read and write. The FPB file is a binary format which is designed to be fast to load. Internally it stores the fingerprints in a way which can be mapped directly to the arena data structure. However, writing this format yourself is not easy.

Instead, let chemfp do it for you. With the *chemfp.open_fingerprint_writer()* function, the difference between writing an FPS file and an FPB file is a matter of changing the extension. Here's a simple example:

```
>>> import chemfp
>>> writer = chemfp.open_fingerprint_writer("simple.fpb")
>>> writer.write_fingerprints( [("first", b"\xff\xff"), ("second", b"ZZ"), ("third", b
↪"\1\2")] )
>>> writer.close()
```

Almost all you need to know is to use the ".fpb" extension instead of ".fps". The rest of this section goes into low-level details that might be enlightening, but probably aren't that directly useful for most people.

It's hard to show the content of the FPB file, because it is binary. I'll do a character dump to show the first 96 bytes:

```
% od -c simple.fpb
0000000    F   P   B   1  \r  \n  \0  \0  \r  \0  \0  \0  \0  \0  \0  \0
0000020    M   E   T   A   #   n   u   m   _   b   i   t   s   =   1   6
0000040   \n   #  \0  \0  \0  \0  \0  \0  \0   A   R   E   N 002  \0  \0
0000060   \0  \b  \0  \0  \0 002  \0  \0 001 002  \0  \0  \0  \0  \0  \0
0000100    Z   Z  \0  \0  \0  \0  \0  \0 377 377  \0  \0  \0  \0  \0  \0
0000120    H  \0  \0  \0  \0  \0  \0  \0   P   O   P   C  \0  \0  \0  \0
  ...
```

The first eight bytes are the file signature. Following that are a set of blocks, with eight bytes for the length, a four byte block type name, and then the block content. Here you can see the "META"data block, followed by the "AREN"a block containing the fingerprint data, followed by the start of the "POPC"ount block with the popcount index information.

That's probably a bit too much detail for you. I'll use chemfp to read the file and show the contents:

```
>>> from __future__ import print_function
>>> import chemfp
>>> reader = chemfp.open("simple.fpb")
>>> print(reader.metadata)
#num_bits=16
```

```
>>> from chemfp import bitops
>>> for id, fp in reader:
...   print(id, "=>", bitops.hex_encode(fp))
...
third => 0102
second => 5a5a
first => ffff
```

Unlike the FPS format, the FPB format requires a *num_bits* in the metadata. Since I didn't give the writer that information, it figured it out from the number of bytes in the first written fingerprint.

You can see that record order is different than the input order. While the FPS fingerprint writer preserves input order, the FPB writer will reorder the records by population count, so the records with fewer 'on' bits come first. It then creates a popcount index, to mark the start and end location of all of the fingerprints with a given popcount. This is used to pre-compute the popcount for a fingerprint, and to implement sublinear similarity search.

Use the *reorder* parameter to control if the fingerprints should be reordered. The default is True, and False will preserve the input order:

```
>>> writer = chemfp.open_fingerprint_writer("simple.fpb", reorder=False)
>>> writer.write_fingerprints( [("first", b"\xff\xff"), ("second", b"ZZ"), ("third", b
↪"\1\2")] )
>>> writer.close()
>>>
>>> reader = chemfp.open("simple.fpb")
>>> for id, fp in reader:
...   print(id, "=>", bitops.hex_encode(fp))
...
first => ffff
second => 5a5a
third => 0102
```

You might think it's a bit useless to preserve input order, because the performance won't be as fast. It's actually proved useful for one project, where the targets were broken up into clusters, and cluster membership was done using a SEA analysis. Rather than have a few dozen separate fingerprint files, I stored everything in the same file (including duplicate fingerprints), and used a configuration file which specified the cluster name and its range in the file. This made it a lot easier to organize the data, and since there were only a few thousand fingerprints sublinear search performance wasn't needed.

The FPB fingerprint writer also has an *alignment* option. If you look very carefully at the character dump you can see that the fingerprints are eight byte aligned:

```
0000040   \n   #  \0  \0  \0  \0  \0  \0  \0   A   R   E   N 002  \0  \0
0000060   \0  \b  \0  \0  \0 002  \0  \0 001 002  \0  \0  \0  \0  \0  \0
0000100    Z   Z  \0  \0  \0  \0  \0  \0 377 377  \0  \0  \0  \0  \0  \0
0000120    H  \0  \0  \0  \0  \0  \0  \0   P   O   P   C  \0  \0  \0  \0
```

The "AREN" is the start of the arena block, the next four bytes ("002 0 0 0 0") are the number of bytes in a fingerprint, in this case 2. The four bytes after that ("b 0 0 0") are the number of bytes allocated for each fingerprint; "b" is the escape code for backspace, or ASCII 8. Yes, 8 bytes are used even though the fingerprints only have 2 bytes in them. This is because the FPB format expects to be able to use the 8 byte "POPC" assembly instruction, if available, because that has the fastest performance.

After the storage size field is a byte for the spacer length. The "002" means two NUL spacer characters follow. This is used to put the start of the first fingerprint on the eight byte boundary, so there will be no alignment issues with using the POPC instruction. (This is not that important for recent Intel processors, but Intel isn't the only processor in the world.)

Finally you see the fingerprints; the first fingerprint is "001 002", followed by six NUL characters to fill up the 8 bytes of storage, the second is "Z Z" followed by six more NUL pad characters, etc.

If you are really working with a two byte fingerprint, then six NUL characters is likely a waste of space. You can ask chemfp to use a two byte alignment instead:

```
>>> import chemfp
>>> writer = chemfp.open_fingerprint_writer("simple.fpb", alignment=2)
>>> writer.write_fingerprints( [("first", b"\xff\xff"), ("second", b"ZZ"), ("third", b
↪"\1\2")] )
>>> writer.close()
```

giving:

```
% od -c simple.fpb
0000000    F   P   B   1  \r  \n  \0  \0  \r  \0  \0  \0  \0  \0  \0  \0
0000020    M   E   T   A   #   n   u   m   _   b   i   t   s   =   1   6
```

```
0000040   \n 017  \0   \0   \0   \0   \0   \0   \0    A    R    E    N 002  \0   \0
0000060   \0 002  \0   \0   \0   \0 001 002    Z    Z 377 377    H   \0   \0   \0
0000100   \0   \0   \0   \0    P    O    P    C   \0   \0   \0   \0   \0   \0   \0   \0
```

If you stare at it long enough you'll see that the storage size is now two bytes, and that the fingerprints are arranged without any padding. (Actually, since chemfp's two byte popcount uses character pointers, you could even use 1 byte alignment without a performance hit. But all this will do is save you at most one byte of spacer.)

Going in the other direction, it's possible to specify up to 256 bytes of alignment. This is far beyond any conceivable use. Even the AVX instructions need only 256 bits, or 32 byte alignment, and that's not a requirement, only a performance optimization to avoid a cache line split.

(If some future instruction set needs a larger alignment then the FPB format acquire a new block type which provides the right alignment.)

Specify the output fingerprint format

In this section you'll learn about the *format* option to the fingerprint writer.

By default *chemfp.open_fingerprint_writer()* uses the destination filename's extension to determine if it should write an FPS file (".fps"), a gzip compressed FPS file (".fps.gz"), or an FPB file (".fpb"). If it doesn't recognize the extension, or if the filename is None (to write to stdout) then it will assume the FPS format.

If the destination is a file-like object then things become a bit more complicated. If the object has a `name` attribute, which is the case with real file objects, then that will be examined for any known extension. That's why the following writes the output in fps.gz format:

```
>>> import chemfp
>>> f = open("example.fps.gz", "wb") # must be in binary mode!
>>> writer = chemfp.open_fingerprint_writer(f)
>>> writer.write_fingerprint("ABC", b"\0\0\0\0")
>>> writer.close()
>>> f.close()
>>> open("example.fps.gz", "rb").read() # must be in binary mode!
"\x1f\x8b\x08\x08\x10%\xdcT\x02\xffexample.fps\x00S ... more deleted
>>>
>>> import gzip
>>> print(gzip.open("example.fps.gz").read())
#FPS1
00000000        ABC
```

Note: Under Python3 that last output will be:

b'#FPS1n00000000tABCn'

There's a large amount of magic behind the scenes to connect the filename in the Python open() call to the chemfp output format.

The other solution is to just tell it which format to use, with the *format* parameter. For example, if you want to send the output to stdout in gzip compressed FPS format then do:

```
writer = chemfp.open_fingerprint_writer(None, format="fps.gz")
```

If you want to save an FPB file to a BytesIO instance then do:

```
from io import BytesIO
f = BytesIO()
writer = chemfp.open_fingerprint_writer(f, format="fpb")
```

And if you really want to save to a file with an ".fpb" extension but have it as an FPS file, then do:

```
writer = chemfp.open_fingerprint_writer("really_an_fps_file.fpb", format="fps")
```

But that would be silly.

Merging multiple structure-based fingerprint sources

In this section you'll learn how to merge multiple fingerprint scores into a single file, and include the full list of source filenames.

The structure-based fingerprint readers include a source filename in the metadata:

```
>>> from __future__ import print_function
>>> import chemfp
>>> filename = "Compound_014550001_014575000.sdf.gz"
>>> reader = chemfp.read_molecule_fingerprints("RDKit-MACCS166", filename)
>>> print(reader.metadata)
#num_bits=166
#type=RDKit-MACCS166/2
#software=RDKit/2017.09.1.dev1 chemfp/3.1
#source=Compound_014550001_014575000.sdf.gz
#date=2017-09-16T00:05:52
```

If you have a single input file and a single output file then you can save the reader to an FPS or FPB file directly:

```
>>> reader.save("example.fpb")
>>> reader.close()
```

Strictly speaking, the `close()` is rarely necessary as the garbage collector will close the file during finalization. Still, it's good practice to close file, and to use a context manager to ensure that the file is always closed. Here's what that looks like:

```
>>> with chemfp.read_molecule_fingerprints("RDKit-MACCS166", filename) as reader:
...     reader.save("example.fpb")
```

However you create it, the output file will have the original metadata:

```
>>> arena = chemfp.open("example.fpb")
>>> print(arena.metadata)
#num_bits=166
#type=RDKit-MACCS166/2
#software=RDKit/2017.09.1.dev1 chemfp/3.1
#source=Compound_014550001_014575000.sdf.gz
```

What happens if you want to want to merge multiple files? How does the output fingerprint file get the correct metadata?

I'll demonstrate the problem by computing fingerprints from two structure files. I'll get the fingerprint type and ask it to create a metadata instance:

```
>>> from __future__ import print_function
>>> import chemfp
>>> filenames = ["Compound_014550001_014575000.sdf.gz", "Compound_027575001_027600000.
↪sdf.gz"]
>>> fptype = chemfp.get_fingerprint_type("RDKit-MACCS166")
>>> print(fptype.get_metadata())
```

```
#num_bits=166
#type=RDKit-MACCS166/2
#software=RDKit/2017.09.1.dev1 chemfp/3.1
#date=2017-09-16T00:07:56
```

The problem is that I also want to include the filenames as source fields in the metadata. The fingerprint type doesn't have this information. Instead, I'll them in through the *sources* parameter, which takes a string or a list of strings:

```
>>> metadata = fptype.get_metadata(sources=filenames)
>>> print(metadata)
#num_bits=166
#type=RDKit-MACCS166/2
#software=RDKit/2017.09.1.dev1 chemfp/3.1
#source=Compound_014550001_014575000.sdf.gz
#source=Compound_027575001_027600000.sdf.gz
#date=2017-09-16T00:08:10
```

What remains is to pass this metadata to the fingerprint writer, then loop through the structure filenames to compute the fingerprints and send them to the writer:

```
>>> with chemfp.open_fingerprint_writer("example.fpb", metadata=metadata) as writer:
...   for filename in filenames:
...     with fptype.read_molecule_fingerprints(filename) as reader:
...       writer.write_fingerprints(reader)
...
```

Here's a quick check to see that the metadata was saved correctly:

```
>>> print(chemfp.open("example.fpb").metadata)
#num_bits=166
#type=RDKit-MACCS166/2
#software=RDKit/2017.09.1.dev1 chemfp/3.1
#source=Compound_014550001_014575000.sdf.gz
#source=Compound_027575001_027600000.sdf.gz
#date=2017-09-16T00:08:10
```

If your toolkit can't parse one of the records then it will raise an exception. You likely want it to ignore errors, which you can do with the *errors* option to *chemfp.read_molecule_fingerprints()*. The final code for this section looks like:

```
import chemfp

filenames = ["Compound_014550001_014575000.sdf.gz", "Compound_027575001_027600000.sdf.
↪gz"]

fptype = chemfp.get_fingerprint_type("RDKit-MACCS166")
metadata = fptype.get_metadata(sources=filenames)

with chemfp.open_fingerprint_writer("example.fpb", metadata=metadata) as writer:
  for filename in filenames:
    with fptype.read_molecule_fingerprints(filename, errors="ignore") as reader:
      writer.write_fingerprints(reader)
```

Merging multiple fingerprint files

In this section you'll learn how to make a modified copy of a metadata instance.

The previous section merged multiple structure-based fingerprints, and used the fingerprint type to get the correct metadata instance.

What if you want to merge several existing fingerprint files, and those use a fingerprint type that chemfp doesn't understand? In that case there is no chemfp fingerprint type, and therefore no *get_metadata()* method to call. Instead, you'll need some other way to make a *chemfp.Metadata* instance.

I'll work through a solution, and start by using *sdf2fps* to extract the PubChem/CACTVS fingerprints from two PubChem SD files:

```
% sdf2fps --pubchem Compound_014550001_014575000.sdf.gz -o Compound_014550001_
↪014575000.fps
% sdf2fps --pubchem Compound_027575001_027600000.sdf.gz -o Compound_027575001_
↪027600000.fps
% head -7 Compound_014550001_014575000.fps | fold
#FPS1
#num_bits=881
#type=CACTVS-E_SCREEN/1.0 extended=2
#software=CACTVS/unknown
#source=Compound_014550001_014575000.sdf.gz
#date=2017-09-16T00:10:01
034e1c0002000000000000000000000000000000000c0000000000000000800000007820201000
003030a51b400d630108421081402442c2004100000444081411006036511 06c444589c9010e0026
0388141be00d03047000020002001000000001000100080000000000000000        14550001
% head -7 Compound_027575001_027600000.fps | fold
#FPS1
#num_bits=881
#type=CACTVS-E_SCREEN/1.0 extended=2
#software=CACTVS/unknown
#source=Compound_027575001_027600000.sdf.gz
#date=2017-09-16T00:10:03
075e1c0002080000000000000000000000000000000c0600000000000000800200007820 0a0000
803510a51b404d93410320501140a44b1a4e430000a450281011980236175 0644c07adb9e18c1026
2b801fd7e919130471000004020020010000000201009000000000000000        27575190
```

Of course you could just ignore the header data, which is what the following does:

```
import chemfp

filenames = ["Compound_014550001_014575000.fps" ,"Compound_027575001_027600000.fps"]

with chemfp.open_fingerprint_writer("merged_pubchem.fps") as writer:
  for filename in filenames:
    with chemfp.open(filename) as reader:
      writer.write_fingerprints(reader)
```

but then you'll be left with no metadata in the FPS header:

```
% head -3 merged_pubchem.fps | fold
#FPS1
034e1c0002000000000000000000000000000000000c0000000000000000800000007820201000
003030a51b400d630108421081402442c2004100000444081411006036511 06c444589c9010e0026
0388141be00d03047000020002001000000001000100080000000000000000        14550001
034e0c0002000000000000000000000000000000000c0000000000000000800000007820081000
003030a51b400d630102401001402442020041000004440810110060361 1106c444589c9010e0026
0b88141be00d03047000020002001000000001000100080000000000000000        14550002
```

While you could do that, the metadata keeps track of potentially useful information, so it's better to add it. For that matter, metadata usually isn't useful until some time after the fingerprints are generated. People tend to put off writing

code until it's needed, but by then it's too late. I've tried to make chemfp's API easy, to encourage people to add the right metadata from the start.

There are a couple of ways to add the right metadata. The classic way is to make your own *chemfp.Metadata* with the right values:

```
>>> metadata = chemfp.Metadata(num_bits=881, type="CACTVS-E_SCREEN/1.0 extended=2",
...   software="CACTVS/unknown", sources=["Compound_014550001_014575000.sdf.gz",
...       "Compound_027575001_027600000.sdf.gz"])
>>> print(metadata)
#num_bits=881
#type=CACTVS-E_SCREEN/1.0 extended=2
#software=CACTVS/unknown
#source=Compound_014550001_014575000.sdf.gz
#source=Compound_027575001_027600000.sdf.gz
```

The downside is this requires knowing all of the fields beforehand. Another option is to *copy* the metadata from the first fingerprint file, and ask the `copy()` to use a new list of sources:

```
>>> from __future__ import print_function
>>> import chemfp
>>> reader = chemfp.open("Compound_014550001_014575000.fps")
>>> metadata = reader.metadata.copy()
>>> metadata.sources
[u'Compound_014550001_014575000.sdf.gz']
>>> metadata = reader.metadata.copy(sources=[
...     u"Compound_014550001_014575000.sdf.gz",
...     u"Compound_027575001_027600000.sdf.gz"])
>>> print(metadata)
#num_bits=881
#type=CACTVS-E_SCREEN/1.0 extended=2
#software=CACTVS/unknown
#source=Compound_014550001_014575000.sdf.gz
#source=Compound_027575001_027600000.sdf.gz
#date=2017-09-16T00:10:01
```

Now to put the pieces together. I'll make one pass through the fingerprint files to get the sources, and then another pass to generate the output. If you only have a handful of files then this works nicely:

```
>>> from __future__ import print_function
>>> import chemfp
>>> filenames = ["Compound_014550001_014575000.fps", "Compound_027575001_027600000.fps
↪"]
>>> readers = [chemfp.open(filename) for filename in filenames]
>>> sources = sum((reader.metadata.sources for reader in readers), [])
>>> sources
[u'Compound_014550001_014575000.sdf.gz', u'Compound_027575001_027600000.sdf.gz']
>>> metadata = readers[0].metadata.copy(sources=sources)
>>> print(metadata)
#num_bits=881
#type=CACTVS-E_SCREEN/1.0 extended=2
#software=CACTVS/unknown
#source=Compound_014550001_014575000.sdf.gz
#source=Compound_027575001_027600000.sdf.gz
#date=2017-09-16T00:10:01

>>> import itertools
>>> with chemfp.open_fingerprint_writer("merged_pubchem.fps", metadata=metadata) as␣
↪writer:
```

```
...     writer.write_fingerprints(itertools.chain.from_iterable(readers))
...
>>> for reader in readers:
...   reader.close()
...
```

(You might not have seen the itertools.chain.from_iterable before. The itertools.chain function iterates over all of the elements in the first term, then the second, etc., as in:

```
>>> list(itertools.chain("abc", "123", "xyz"))
['a', 'b', 'c', '1', '2', '3', 'x', 'y', 'z']
```

The itertools.chain.from_iterable takes an iterable, like a list, as its sole parameter:

```
>>> list(itertools.chain.from_iterable(["abc", "123", "xyz"]))
['a', 'b', 'c', '1', '2', '3', 'x', 'y', 'z']
```

)

However, the previous code likely won't work if you want to merge thousands of records, which might happen if you try to merge all of the PubChem file. Why? Because the operating system may limit the number of open file handles:

```
>>> import glob
>>> filenames = glob.glob("/Users/dalke/databases/pubchem/*.sdf.gz")
>>> filenames = filenames + filenames # double the size to reach my system limit
>>> readers = [open(filename) for filename in filenames]
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
IOError: [Errno 24] Too many open files: '/Users/dalke/databases/pubchem/Compound_
↪029800001_029825000.sdf.gz'
>>> filenames.index("/Users/dalke/databases/pubchem/Compound_029800001_029825000.sdf.
↪gz")
1192
>>> filenames.index("/Users/dalke/databases/pubchem/Compound_029800001_029825000.sdf.
↪gz", 1193)
4861
```

Double-checking with Python's resource module:

```
>>> import resource
>>> resource.getrlimit(resource.RLIMIT_NOFILE)
(4864, 9223372036854775807)
```

This says that there's a "soft" limit of 4864 open files, though I could change that to a larger number, up to the much higher "hard" limit of roughly 9 quintillion. What happened in the above exception was that the 4862nd file reached the soft limit. Remember, stdin, stdout, and stderr are also open files, and 4861+3 = 4864, which was the limit.

I did have to cheat in the above to get the exception. I doubled the list of filenames. The original version of this documentation was written on a version of the Mac operating system which had a default soft limit of only 256 open files. It was easy to reach the limit with just the list of PubChem filenames.

Even if you only think you'll open a few dozen files, you might want to write code which doesn't tempt the limits. The following will do one scan through the files and create a *Metadata* instance with all of the sources from each of the files. I'll use the `with` statement to automatically close the file during this scan. I'll then do another pass through the filenames to merge all of the fingerprints into a single file:

```
import chemfp
filenames = ["Compound_014550001_014575000.fps", "Compound_027575001_027600000.fps"]
```

```
# Create the correct metadata with all of the sources from all of the files.
metadata = None
sources = []
for filename in filenames:
    with chemfp.open(filename) as reader:
        if metadata is None:
            metadata = reader.metadata.copy()
        sources.extend(reader.metadata.sources)

metadata = metadata.copy(sources = sources)

# Merge the files using the new metadata
with chemfp.open_fingerprint_writer("merged_pubchem.fps", metadata=metadata) as␣
↪writer:
    for filename in filenames:
        with chemfp.open(filename) as reader:
            writer.write_fingerprints(reader)
```

This code assumes that the fingerprints are compatible, that is, that the fingerprints are the same size, and the fingerprint types and other metadata fields are compatible. The next section shows how to detect if there are compatibility problems.

Check for metadata compatibility problems

In this section you'll learn how to detect compatibility mismatches between two metadata instances, and between a metadata and a fingerprint.

In the previous section you learned how to merge multiple fingerprint files, which all happened to have the same fingerprint type. What happens if they are different types?

There are actually a few possible problems:

- the fingerprint lengths are different (very bad)
- the fingerprint types are different (probably bad)
- the software is from different versions (probably okay)

The *chemfp.check_metadata_problems()* function compares two metadata objects and returns a list of possible problems:

```
>>> from __future__ import print_function
>>> import chemfp
>>> rdkit_metadata = chemfp.get_fingerprint_type("RDKit-MACCS166").get_metadata()
>>> openeye_metadata = chemfp.get_fingerprint_type("OpenEye-MACCS166").get_metadata()
>>> problems = chemfp.check_metadata_problems(rdkit_metadata, openeye_metadata)
>>> len(problems)
2
>>> for problem in problems:
...   print(problem)
...
WARNING: query has fingerprints of type 'RDKit-MACCS166/2' but
target has fingerprints of type 'OpenEye-MACCS166/3'
INFO: query comes from software 'RDKit/2017.09.1.dev1 chemfp/3.1'
but target comes from software 'OEGraphSim/2.2.6 (20170208) chemfp/3.1'
```

In this case the fingerprint types are different, but since the fingerprint lengths are the same it's not an error, only a warning. The software field is also not identical, but as that's not so significant it's listed as "info".

The returned problem objects are *chemfp.ChemFPProblem()* instances, which have useful attributes:

```
>>> for problem in problems:
...   print("Problem:")
...   print("  severity:", problem.severity)
...   print("  category:", problem.category)
...   print("  description:", problem.description)
...
Problem:
  severity: warning
  category: type mismatch
  description: query has fingerprints of type 'RDKit-MACCS166/2'
     but target has fingerprints of type 'OpenEye-MACCS166/3'
Problem:
  severity: info
  category: software mismatch
  description: query comes from software 'RDKit/2017.09.1.dev1 chemfp/3.1'
     but target comes from software 'OEGraphSim/2.3.1.b.2_debug (20170828) chemfp/3.1'
```

The idea is that the `category` text won't change, so your code can figure out what's going on, while the `description` is subject to change and hopefully improvement. The severity is one of "info", "warning" and "error".

```
>>> rdkit1_metadata = chemfp.get_fingerprint_type("RDKit-Fingerprint fpSize=512").get_
↪metadata()
>>> rdkit2_metadata = chemfp.get_fingerprint_type("RDKit-Fingerprint fpSize=1024").
↪get_metadata()
>>> problems = chemfp.check_metadata_problems(rdkit1_metadata, rdkit2_metadata)
>>> for problem in problems:
...   print(problem)
...
ERROR: query has 512 bit fingerprints but target has 1024 bit fingerprints
WARNING: query has fingerprints of type 'RDKit-Fingerprint/2 minPath=1
maxPath=7 fpSize=512 nBitsPerHash=2 useHs=1' but target has
fingerprints of type 'RDKit-Fingerprint/2 minPath=1 maxPath=7 fpSize=1024
nBitsPerHash=2 useHs=1'
```

A *chemfp.ChemFPProblem* is derived from `Exception`, so you can raise it directly if you want:

```
>>> for problem in chemfp.check_metadata_problems(rdkit1_metadata, rdkit2_metadata):
...   if problem.severity == "error":
...     raise problem
...
Traceback (most recent call last):
  File "<stdin>", line 3, in <module>
chemfp.ChemFPProblem: ERROR: query has 512 bit fingerprints but target has 1024 bit␣
↪fingerprints
```

You might have noticed that the error message uses the words "query" and "target". Chemfp is designed around similarity searches, so I expect the default to compare query metadata to target metadata.

On the other hand, the previous section merged multiple fingerprint files, where "query" and "target" don't make sense. Instead, you can give alternative names via the *query_name* and *target_name* parameters:

```
>>> rdkit1_metadata = chemfp.get_fingerprint_type("RDKit-Fingerprint fpSize=512").get_
↪metadata()
>>> rdkit2_metadata = chemfp.get_fingerprint_type("RDKit-Fingerprint fpSize=1024").
↪get_metadata()
```

```
>>> for problem in chemfp.check_metadata_problems(rdkit1_metadata, rdkit2_metadata,
...                                               "file #1", "file #14"):
...   if problem.severity == "error":
...     print(problem)
...
ERROR: file #1 has 512 bit fingerprints but file #14 has 1024 bit fingerprints
```

I'll use this to update the code from the previous section to raise an exception on errors, print warnings to stderr, and do nothing about "info" problems, and add a MACCS fingerprint file to the list of files to process, so I can show what happens if there's a problem:

```
import sys
import chemfp

filenames = ["Compound_014550001_014575000.fps",
             "Compound_027575001_027600000.fps",
             "chebi_maccs.fps"]

# Create the correct metadata with all of the sources from all of the files.
metadata = None
sources = []
for filename in filenames:
    with chemfp.open(filename) as reader:
        if metadata is None:
            metadata = reader.metadata.copy()
            first_filename = filename
        else:
            # Check for compatibility problems
            for problem in chemfp.check_metadata_problems(metadata, reader.metadata,
                                                          repr(first_filename),
                                                          repr(filename)):
                if problem.severity == "error":
                    raise problem
                elif problem.severity == "warning":
                    sys.stderr.write(str(problem) + "\n")

        sources.extend(reader.metadata.sources)

if metadata is not None:
  metadata = metadata.copy(sources=sources)

# Merge the files using the new metadata
with chemfp.open_fingerprint_writer("merged_pubchem.fps", metadata=metadata) as␣
↪writer:
    for filename in filenames:
        with chemfp.open(filename) as reader:
            writer.write_fingerprints(reader)
```

When I run that code with the mismatched fingerprint types, I get the error message:

```
Traceback (most recent call last):
  File "x.py", line 23, in <module>
    raise problem
chemfp.ChemFPProblem: ERROR: 'Compound_014550001_014575000.fps' has
881 bit fingerprints but 'chebi_maccs.fps' has 166 bit fingerprints
```

I then removed the `chebi_maccs.fps` and manually changed the fingerprint type, so I could demonstrate what a warning message looks like:

```
WARNING: 'Compound_014550001_014575000.fps' has fingerprints of type
u'CACTVS-E_SCREEN/1.0 extended=2' but
'Compound_027575001_027600000.fps' has fingerprints of type
u'CACTVS-E_SCREEN/1.0 extended=DIFFERENT_VALUE'
```

(In case you're wondering what the type string means, those are the actual CACTVS parameters that PubChem uses, according to the CACTVS author, Wolf-Dietrich Ihlenfeldt.)

Lastly, sometimes the query is a simple byte string. There's not really much to compare, but you use *chemfp.check_fingerprint_problems()* to see if the fingerprint length is compatible with a metadata instance:

```
>>> import chemfp
>>> metadata = chemfp.get_fingerprint_type("RDKit-MACCS166").get_metadata()
>>> chemfp.check_fingerprint_problems(b"\0\0\0\0", metadata)
[ChemFPProblem('error', 'num_bytes mismatch', 'query contains 4
bytes but target has 21 byte fingerprints')]
```

The *simsearch* command-line tool uses this function to check if the query fingerprint, which is entered as hex as a command-line parameter, is compatible with the target fingerprints.

How to write very large FPB files

In this section you'll learn how to write an FPB file even when fingerprint data is so large that the intermediate data doesn't all fit into memory at once.

By default the FPB format will reorder the fingerprints to be in popcount order. (Use `reorder=False` option to preserve the input order.) This requires intermediate storage in order to sort all of the records. By default the writer will use memory for this, but the implementation may require about two to three times as much memory as the raw fingerprint size.

That is, if you have 50 million fingerprints, with 1024 bits per fingerprint, plus 10 bytes for the name, then the fingerprint arena requires about 6 GiB of memory, plus 0.5 GiB for the ids, and another ~1 GiB for the id lookup table.

That calculation gives the minimum amount of memory needed. The actual implementation may preallocate up to twice as much memory as the current size, in order to handle growth gracefully, and there is some additional overhead. You may be left with the case where you have 12 GiB of RAM, and where the final FPB file is only 8 GiB in size, but where the intermediate storage requires 15 GiB of RAM.

Or you may want to build that data set on a machine with 6 GiB of RAM, and copy the result over to the production machine with a lot more memory.

If that happens, then use the *max_spool_size* option to specify the maximum number of bytes to store in memory before switching to temporary files for additional storage. This should be about 1/3 of the available RAM because there can be two different temporary file "spools", each of which can use up to *max_spool_size* bytes of RAM.

For example, the following will use at most about 4 GiB of RAM:

```
writer = chemfp.open_fingerprint_writer(
    "pubchem.fpb", max_spool_size = 2 * 1024 * 1024 * 1024)
```

Note: do not make this too small. The merge step opens all of the temporary files in order to make the final FPB output file. If you specify a spool size of 50 MiB then you'll end up creating several hundred files for PubChem, which may exceed the resource limits for the number of open file descriptors for a process. When that happens you'll get an exception like:

```
IOError: [Errno 24] Too many open files
```

Where does the FPB writer store the temporary files? It uses Python's tempfile module to create the temporary files in a directory. Quoting from that documentation, "The default directory is chosen from a platform-dependent list, but the user of the application can control the directory location by setting the TMPDIR, TEMP or TMP environment variables."

Environment variables give one way to specify an alternate directory. Or you can specify it directly using the *tmpdir* parameter, as in:

```
writer = chemfp.open_fingerprint_writer(
    "pubchem.fpb", max_spool_size = 2 * 1024 * 1024 * 1024,
    tmpdir = "/scratch")
```

This can be very important on some cluster machines with a small local /tmp but a large networked scratch disk.

FPS fingerprint writer errors

In this section you'll learn how the FPS fingerprint writer handles errors, and how to change the error handling behavior.

It's hard but not impossible to have the FPS writer raise an exception:

```
>>> import chemfp
>>> writer = chemfp.open_fingerprint_writer(None)
#FPS1
>>> writer.write_fingerprint("Tab\tHere", b"\0")
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "chemfp/fps_io.py", line 550, in write_fingerprint
    raise_tb(err[0], err[1])
  File "chemfp/fps_io.py", line 467, in _fps_writer_gen
    location)
  File "chemfp/io.py", line 87, in error
    _compat.raise_tb(ParseError(msg, location), None)
  File "<string>", line 1, in raise_tb
chemfp.ParseError: Unable to write an identifier containing a tab: 'Tab\tHere', file '
↪<stdout>', line 1, record #1
```

The FPS file format simply doesn't support tab characters in the indentifier, nor newline characters, for that matter. It also doesn't allow empty identifiers.

As you saw, the default error action is to raise an exception.

Sometimes it's okay to ignore errors. For example, you might process a large number of structures, where you know that a few of them have missing, or poorly formed, identifiers, and where it's okay to omit those records.

The *errors* parameter can be used to change the error handler. The value of "report" tells the parser to skip failing record and write an error message written to stderr. The value of "ignore" simply skips the record:

```
>>> writer = chemfp.open_fingerprint_writer(None, errors="report")
#FPS1
>>> writer.write_fingerprint("", b"\0\0\0\0")
ERROR: Unable to write a fingerprint with an empty identifier, file '<stdout>', line␣
↪1, record #1. Skipping.
>>>
>>> writer = chemfp.open_fingerprint_writer(None, errors="ignore")
#FPS1
>>> writer.write_fingerprint("", b"\0")
>>> writer.write_fingerprint("Tab\tHere", b"\0")
```

Granted, this feature isn't so important for *FingerprintWriter.write_fingerprint()* because catching the exception isn't hard to do. It's a bit more useful for bulk conversions with *FingerprintWriter.write_fingerprints()*, like:

```
import chemfp
with chemfp.read_molecule_fingerprints("RDKit-MACCS166", "Compound_014550001_
↪014575000.sdf.gz") as reader:
  with chemfp.open_fingerprint_writer("example.fps", reader.metadata, errors="report
↪") as writer:
    writer.write_fingerprints(reader)
```

Note that the FPB writer ignores the *errors* parameter and treats all errors as "strict".

FPS fingerprint writer location

In this section you'll learn how to get information like the number of lines and number of records written to an FPS file.

I'll start by saying that this feature isn't all that useful. It exists because of parallelism to the toolkit structure writers, and I wanted to experiment to see if it could be useful in the future.

The *FPS fingerprint writer* has a `location` attribute. This can be used to get some information about the state of the output writer. The most basic is the output filename. If the output is None or an unnamed file object then a fake filename will be used:

```
>>> import chemfp
>>> writer = chemfp.open_fingerprint_writer("example.fps")
>>> writer.location.filename
'example.fps'
>>> writer = chemfp.open_fingerprint_writer(None)
#FPS1
>>> writer.location.filename
'<stdout>'
```

At this point the signature line has been written, so the file is at line 1, but no record have been written:

```
>>> writer.location.lineno
1
>>> writer.location.recno
0
>>> writer.location.output_recno
0
```

Each of these values is incremented by one after adding a valid record:

```
>>> writer.write_fingerprint("FP001", b"\xA0\xFE")
a0fe    FP001
>>> writer.location.lineno
2
>>> writer.location.recno
1
>>> writer.location.output_recno
1
```

If however the record is invalid then the `recno` will increase by one because it's the number of records sent to the writer, but the other values do not increase because they only change when a record is written successfully:

```
>>> writer.write_fingerprint("", b"\xA0\xFE")
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "chemfp/fps_io.py", line 550, in write_fingerprint
    raise_tb(err[0], err[1])
  File "chemfp/fps_io.py", line 475, in _fps_writer_gen
    location)
  File "chemfp/io.py", line 87, in error
    _compat.raise_tb(ParseError(msg, location), None)
  File "<string>", line 1, in raise_tb
chemfp.ParseError: Unable to write a fingerprint with an empty identifier, file '
↪<stdout>', line 2, record #2
>>> writer.location.lineno
2
>>> writer.location.recno
2
>>> writer.location.output_recno
1
```

This is perhaps more clearly shown if I try to write four records at one, where two contain errors, and where I've asked the writer to "report" errors rather than raise an exception:

```
>>> metadata = chemfp.Metadata(type="Experiment/1", software="AndrewDalke/1")
>>> writer = chemfp.open_fingerprint_writer(None, metadata=metadata, errors="report")
#FPS1
#type=Experiment/1
#software=AndrewDalke/1
>>> writer.location.lineno
3
>>> writer.location.recno
0
>>> writer.location.output_recno
0
>>> writer.write_fingerprints( [("A", b"\0\0"), ("\t", b"\0\1"), ("", b"\0\2"), ("B",␣
↪b"\0\3")] )
0000  A
ERROR: Unable to write an identifier containing a tab: '\t', file '<stdout>', line 4,␣
↪record #2. Skipping.
ERROR: Unable to write a fingerprint with an empty identifier, file '<stdout>', line␣
↪4, record #3. Skipping.
0003  B
>>> writer.location.recno
4
>>> writer.location.output_recno
2
>>> writer.location.lineno
5
```

There are three lines in the header; the signature, the type line, and the software line. I tried to write four fingerprints, but two were invalid. It wrote the valid fingerprint "A" to stdout, report the two invalid records to stderr, and write the valid fingerprint "B" to stdout. Thus, two records were actually output, which is why `output_recno` is 2, while four records were sent to the writer, which is why `recno` is 4. The three header lines and the two lines of output give five lines of output, so the final `lineno` is 5.

In case you hadn't figured it out, the location information is used to make the exception and error message. That explains why both of the error reports say the error is on "line 4"; that's the line that would have been output if there were no error.

Note that the `FPB writer` does not have a location, and it ignores the `location` parameter.

MACCS dependency on hydrogens

In this section you'll learn how the RDKit MACCS fingerprints differ if there are explicit or implicit hydrogens.

Note: A goal of this is to show that MACCS key generation isn't as easy as you might think it is!

One of my long-term goals is to get a good cross-toolkit implementation of the MACCS keys. It's very odd how the MACCS keys are the de facto fingerprint for cheminformatics, but the toolkits don't give the same answers. Over the years, I've found bugs or incomplete definitions in all of the toolkits I've looked at, which I've reported and have since been fixed.

If you use RDKit, Open Babel, or CDK (chemfp doesn't yet support CDK, but this is my story so I get to mention it) then your toolkit implements MACCS keys that were derived from the ones that Greg Landrum developed for RDKit. The portable portion uses hand-translated SMARTS definitions for most of the MACCS key definitions. A couple keys, like key 125 ("at least two aromatic rings") cannot be represented as SMARTS. RDKit had special code for these definitions, but Open Babel does not.

Even with a portable SMARTS definition, I would expect to see some differences between the toolkits, if only because they have different aromaticity models. One toolkit might call something an aromatic ring, while another says it's aliphatic.

Unfortunately, the SMARTS patterns used in those programs give different results if you have explicit hydrogens or implicit hydrogens. I'll demonstrate with using RDKit, because that has a *reader_arg* to specify if I want to remove hydrogens from the input structure or not. (Here "remove" means to make them implicit.)

I'll use RDKit twice to read the first molecule from a file and compute the RDKit fingerprint; the first time I keep the hydrogens and the second time I remove them:

```
>>> import chemfp
>>> from chemfp import bitops
>>> filename = "Compound_014550001_014575000.sdf.gz"
>>> fptype = chemfp.get_fingerprint_type("RDKit-MACCS166")
>>>
>>> with_h_reader = fptype.read_molecule_fingerprints(filename,
...    reader_args={"removeHs": False})
>>> with_h_id, with_h_fp = next(with_h_reader)
>>> with_h_id, bitops.hex_encode(with_h_fp)
('14550001', '000080000000081406000a326a090616a5fceae7d1f')
>>>
>>> without_h_reader = fptype.read_molecule_fingerprints(filename,
...    reader_args={"removeHs": True})
>>> without_h_id, without_h_fp = next(without_h_reader)
>>> without_h_id, bitops.hex_encode(without_h_fp)
('14550001', '000080000000081406000a226a010614a5fceae7d1f')
```

If you look closely you'll see that they have two different fingerprints! I'll make it easier to see by reporting the bits which are only in one or the other fingerprint:

```
>>> with_h_bits = set(bitops.byte_to_bitlist(with_h_fp))
>>> without_h_bits = set(bitops.byte_to_bitlist(without_h_fp))
>>> sorted(with_h_bits - without_h_bits) # only with hydrogens
[80, 111, 125]
>>> sorted(without_h_bits - with_h_bits) # only without hydrogens
[]
```

The molecule with explicit hydrogens sets three more bits than the one with implicit hydrogens.

Why is that? The RDKit (and hence Open Babel and CDK) definitions often use "*" to match an atom, when the corresponding MACCS definition is supposed to exclude hydrogens. A hydrogen-independent version would use "[!#1]" instead. By default RDKit removes normal explicit hydrogens, so this isn't usually a problem. As far as I can tell, Open Babel always removes them from an SD file, so again this isn't really a problem. (Well, except for hydrogens with an explicit isotope number.)

The list [80, 111, 125] are bit numbers. The corresponding keys are [81, 112, 126]. I looked at how those are defined in various sources:

```
Definitions for key 112 (bit 111)
    MACCS: AA(A)(A)A
    RDKit:  *~*(~*)(~*)~*
    OpenBabel: *~*(~*)(~*)~*
    CDK: *~*(~*)(~*)~*
    chemfp's RDMACCS-*: [!#1]~*(~[!#1])(~[!#1])~[!#1]
    O'Donnell: *~*(~*)(~*)~*
```

("O'Donnell" here comes from Table A.4 of TJ O'Donnell's Design and Use of Relational Databases in Chemistry.)

If you know SMARTS you can see how an explicit H will lead to a different match than an implicit one, except for chemfp's own attempt at making a cross-toolkit MACCS implementation. I'll test out RDMACCS-RDKit, which is chemfp's implementation of the MACCS 166 fingerprint using RDKit:

```
>>> chemfp_maccs = chemfp.get_fingerprint_type("RDMACCS-RDKit")
>>>
>>> with_h_reader = chemfp_maccs.read_molecule_fingerprints(filename,
...     reader_args={"removeHs": False})
>>> with_h_id, with_h_fp = next(with_h_reader)
>>> with_h_id, bitops.hex_encode(with_h_fp)
('14550001', '000080000000081406000a226a010614a5fceae7d1f')
>>>
>>> without_h_reader = chemfp_maccs.read_molecule_fingerprints(filename,
...     reader_args={"removeHs": True})
>>> without_h_id, without_h_fp = next(without_h_reader)
>>> without_h_id, bitops.hex_encode(without_h_fp)
('14550001', '000080000000081406000a226a010614a5fceae7d1f')
>>>
>>> with_h_fp == without_h_fp
True
```

What a relief that they are the same!

If you want to use the OEChem or Open Babel-based RDMACSS implemenations, the corresponding fingerprint type names are "RDMACCS-OpenEye" or "RDMACCS-OpenBabel", respectively, and the command-line option for *oe2fps* and *ob2fps* is `--rdmaccs`.

WARNING: the RDMACCS fingerprints have not been fully validated! Validation is hard. A chemfp goal is to make that easier.

To finish, I was curious about the differences in RDKit's native MACCS166 implementation across all of the records in the file, so I wrote some code. It's a direct evolution of the code you already saw. The only new things might be that izip function from the itertools module in Python 2. It's the same as `zip`, except where zip returns the complete list, izip returns an iterator for elements that would be in that list. A simplifed implementation looks like:

```
def izip(iter1, iter2):
    while 1:
        yield next(iter1), next(iter2)
```

This changed in Python 3. The built-in `zip` now returns an iterator instead of a list, and `itertools.izip` has been

removed. I want the following code to work under Python 2 and Python 3, so I wrote a portability layer to refer to the appropriate function as `izip`:

```
# Python 2
>>> import itertools
>>> izip = getattr(itertools, "izip", zip)
>>> izip is zip
False
>>> izip is itertools.izip
True
# Python 3
>>> import itertools
>>> izip = getattr(itertools, "izip", zip)
>>> izip is zip
True
```

Otherwise, I think I've covered what's needed to understand the rest of the code without more elaboration:

```
from __future__ import print_function
import itertools
from collections import Counter
import chemfp
from chemfp import bitops

izip = getattr(itertools, "izip", zip) # Support Python2 and Python3

filename = "Compound_014550001_014575000.sdf.gz"
with_h_fingerprints = chemfp.read_molecule_fingerprints(
    "RDKit-MACCS166", filename, reader_args={"removeHs": False})
without_h_fingerprints = chemfp.read_molecule_fingerprints(
    "RDKit-MACCS166", filename, reader_args={"removeHs": True})

extra_with_h = Counter()
extra_without_h = Counter()
num_records = 0
for (id1, with_h_fp), (id2, without_h_fp) in izip(with_h_fingerprints,
                                                  without_h_fingerprints):
    num_records += 1
    assert id1 == id2, (id1, id2)
    if with_h_fp != without_h_fp:
        with_h_keys = set(bitno+1 for bitno in bitops.byte_to_bitlist(with_h_fp))
        without_h_keys = set(bitno+1 for bitno in bitops.byte_to_bitlist(without_h_
↪fp))
        only_with_h = sorted(with_h_keys - without_h_keys)
        only_without_h = sorted(without_h_keys - with_h_keys)
        print(id1, "with:", only_with_h, "without:", only_without_h)
        extra_with_h.update(only_with_h)
        extra_without_h.update(only_without_h)

print("\nNumber of records:", num_records)
print("\nCounts that were only with hydrogens:")
for key, count in extra_with_h.most_common():
    print("  %d %d" % (key, count))
print("\nCounts that were only without hydrogens:")
for key, count in extra_without_h.most_common():
    print("  %d %d" % (key, count))
```

In case you were wondering, the report summary starts:

```
Number of records: 5167

Counts that were only with hydrogens:
  112 2993
  150 2116
  144 1939
  138 1283
  122 1201
```

Now you can see why I used key 112 in my elaboration - it's the one that causes the most problems!

Create similarity search web service

In this section you'll learn how to write a simple WSGI-based web service which does a similarity search given an SDF record.

I found it a bit difficult to write this section because few people will write a WSGI service directly. I think most people use Django, but a Django example would require several different files to make it work. There are other web frameworks I could use, like Flask, but I eventually decided to limit myself to what's available in the standard library, that is, the wsgiref module.

I'm going to write a WSGI server named "simple_server.py" which takes an SDF record as input and returns the top 5 hits from a specified database. If there's a GET request then the result is a simple form. The form sends a POST request to the server, with the SDF record in the query parameter *q*.

By the way, if the target fingerprint data set is large then you should use an FPB file to get the best startup performance.

Let's get started. The first part is a comment about what the code does and some imports:

> # This is a very simple fingerprint search server. # I call it 'simple_server.py'. # # Usage: simple_server <fingerprint_filename> [port] # # A GET to the server (default uses port 80) returns a simple form. # The form has a single text box, to paste the SDF query or queries. # The POST query variable 'q' contains the SDF contents. # The search finds the nearest 5 queries for each query record. # The result is a simple list of query ids and its matches.
>
> import argparse from wsgiref.simple_server import make_server import cgi
>
> import chemfp

The server will return an HTML form for a GET request:

```
# Create a simple form.
def query_form(environ, start_response):
    status = '200 OK' # HTTP Status
    headers = [('Content-type', 'text/html')] # HTTP Headers
    start_response(status, headers)

    # The returned object is going to be printed.
    # Must be a byte string for Python 3.
    return [b"""<html>
<head>
 <title>Simple fingerprint search</title>
</head>
<body>
<form method="POST">
Paste in SDF records(s):<br />
<textarea name="q" type="text" rows="20" cols="80"></textarea><br />
<button type="submit">Search!</button>
</form>
```

```
</body>
</html>
"""]
```

I'll use the argparse module to handle the command-line arguments:

```
# Command-line parameters
parser = argparse.ArgumentParser("simple_search",
                                 description="Simple fingerprint web server with SDF␣
↪input")
parser.add_argument("filename",
                    help="chemfp fingerprint filename")

parser.add_argument("port", type=int, default=8080, nargs="?",
                    help="port to use (default is 8080)")
```

The heavy work is in the 'main' function. It starts with some setup to load the fingerprints and make sure the fingerprint type is available:

```
def main():
    args = parser.parse_args()

    # Load the arena, get the type, and make sure I can handle the type.
    arena = chemfp.load_fingerprints(args.filename)
    print("Loaded %s fingerprints from %r" % (len(arena), args.filename))

    type = arena.metadata.type
    if type is None:
        parser.error("File %r does not contain a fingerprint type" % (args.filename,))

    try:
        fptype = chemfp.get_fingerprint_type(type)
    except KeyError as err:
        parser.error(str(err))
```

It then defines the WSGI app, which returns the query_form() for a GET request, or processes the form for a POST request. I think the embedded comments explain things enough:

```
# ... continue the 'main' function ...
  # This is the WSGI app

  def fingerprint_search_app(environ, start_response):
      # Is this a GET or a POST? If a GET, return the query form
      if environ["REQUEST_METHOD"] != "POST":
          return query_form(environ, start_response)

      # Get the query data from the POST
      post_env = environ.copy()
      post = cgi.FieldStorage(
          fp=environ['wsgi.input'],
          environ=post_env,
          keep_blank_values=True,
      )
      q = post.getfirst("q", "")
      # The underlying toolkit code may require "\n" instead of "\r\n" strings.
      q = q.replace("\r\n", "\n")

      # For each input record, do a search, get the results, and build up the output␣
↪lines.
```

```
      # Ignore any records that can't be parsed.

      output = ["Search against %r using k=5 and threshold=0.0\n\n" % (args.filename,
↪)]

      # The next three lines use chemfp to convert the record into a
      # fingerprint, do the search for the top 5 hits, get the ids
      # and scores for the hits, and make the output text.

      for query_id, fp in fptype.read_molecule_fingerprints_from_string(q, "sdf",␣
↪errors="ignore"):
          results = arena.knearest_tanimoto_search_fp(fp, k=5, threshold=0.0)
          text = " ".join( "%s (%.3f)" % (id, score) for (id, score) in results.get_
↪ids_and_scores())
          output.append("%s => %s\n" % (query_id, text))

      # Return the results in plain text.

      status = '200 OK' # HTTP Status
      headers = [('Content-type', 'text/plain')] # HTTP Headers
      start_response(status, headers)

      # Python 3 requires bytes, not strings, so convert to UTF-8
      return output
```

The main function ends with some code to start the WSGI server using the correct port:

```
# ... end of the 'main' function ...
  # Make the server and run it. (Use ^C to kill it.)
  httpd = make_server('', args.port, fingerprint_search_app)
  print("Serving fingerprint search on port %s..." % (args.port,))

  httpd.serve_forever()
```

Finally, code to start things rolling:

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

I'll start the server using a ChEBI-derived data set:

```
% python simple_server.py rdkit_chebi.fps
Loaded 93572 fingerprints from 'rdkit_chebi.fps'
Serving fingerprint search on port 8080...
```

then direct the browser to http://127.0.0.1:8080/ . I pasted in the first three records from ChEBI itself, pressed "Search!", and got the result:

```
Search against 'rdkit_chebi.fps' using k=5 and threshold=0.0

CHEBI:776 => CHEBI:776 (1.000) CHEBI:87628 (1.000) CHEBI:34165 (0.906) CHEBI:17263 (0.
↪886) CHEBI:91668 (0.886)
CHEBI:1148 => CHEBI:1148 (1.000) CHEBI:50612 (1.000) CHEBI:50613 (1.000) CHEBI:64552␣
↪(1.000) CHEBI:73709 (1.000)
CHEBI:1734 => CHEBI:1734 (1.000) CHEBI:18088 (0.916) CHEBI:77688 (0.916) CHEBI:18220␣
↪(0.896) CHEBI:29608 (0.883)
```

I don't think I'll continue this WSGI example in future documentation as that API is too low-level and seldom used

by web developers. If you think otherwise, let me know.

Fingerprint family and type examples

This chapter describes how to use the fingerprint family and fingerprint type API added in chemfp 2.0.

Fingerprint families and types

In this section you'll learn the difference between a fingerprint family and a fingerprint type. You will need Compound_014550001_014575000.sdf.gz from PubChem to work though all of the examples.

Chemfp distinguishes between a "fingerprint family" and a "fingerprint type." A fingerprint family describes the general approach for doing a fingerprint, like "the OpenEye path-based fingerprint method", while a fingerprint type describes the specific parameters used for a given approach, such as "the OpenEye path-based fingerprint method using path lengths between 0 and 5 bonds, where the atom types are based on the atomic number and aromaticity, and the bond type is based on the bond order, mapped to a 256 bit fingerprint."

(In object-oriented terms, a fingerprint family is the class and a fingerprint type is an instance of the class.)

I'll use *chemfp.get_fingerprint_family()* to get the *FingerprintFamily* for "OpenEye-Path". On the laptop where I'm writing the documentation, this resolves to what chemfp calls version "2":

```
>>> from __future__ import print_function
>>> import chemfp
>>> family = chemfp.get_fingerprint_family("OpenEye-Path")
>>> family
FingerprintFamily(<OpenEye-Path/2>)
```

The fingerprint family can be called like a function to return a *FingerprintType*. If you call it with no arguments it will use the defaults parameters for that family. I'll do that, then use *get_type()* to get the fingerprint type string, which is the canonical representation of the fingerprint family name, version, and parameters:

```
>>> fptype = family()
>>> fptype.get_type()
'OpenEye-Path/2 numbits=4096 minbonds=0 maxbonds=5␣
↪atype=Arom|AtmNum|Chiral|EqHalo|FCharge|HvyDeg|Hyb btype=Order|Chiral'
```

A 4096 bit fingerprint is rather large. I'll make a new OpenEye-Path fingerprint type, but this time with only 256 bits. That's small enough that the resulting fingerprint will fit on a line of documentation. All of the other parameters will be unchanged:

```
>>> fptype = family(numbits=256)
>>> fptype
<chemfp.openeye_types.OpenEyePathFingerprintType_v2 object at 0x10b9c4e90>
>>> print(fptype.get_metadata())
#num_bits=256
#type=OpenEye-Path/2 numbits=256 minbonds=0 maxbonds=5␣
↪atype=Arom|AtmNum|Chiral|EqHalo|FCharge|HvyDeg|Hyb btype=Order|Chiral
#software=OEGraphSim/2.2.6 (20170208) chemfp/3.1
#date=2017-09-16T13:56:20
```

This time I used *FingerprintType.get_metadata()* to give information about the fingerprint. This returns a new *Metadata* instance which describes the fingerprint type, and if you print a Metadata it displays the metadata information as an FPS header.

Once you have the fingerprint type you can create fingerprints, including directly from a SMILES string, as in the following:

```
>>> from chemfp import bitops
>>> fp = fptype.parse_molecule_fingerprint("c1ccccc1O", "smistring")
>>> bitops.hex_encode(fp)
'00122501609010000800028100004002010009000548804420000e8040201000'
```

and from a structure file:

```
>>> for id, fp in fptype.read_molecule_fingerprints("Compound_014550001_014575000.sdf.
↪gz"):
...   print(id, bitops.hex_encode(fp))
...
14550001 5ae8f4bbfcda6a66fdbfc2ab9045ecde36b055e3ca56f10477a18df6fd1ebb06
14550002 5ac8f4fafcce6b657d3f82a79145aacca65015e34a56c00777880db27d8ef006
14550003 78c8f17a7cce6b657d3782a59105a2c4a64115c34a5ec04773a80fb2758cd006
14550004 2683e056c28a20882ba8d410304184514213c0300209c3e0eb8241b280008102
         ...
```

For more examples of using `get_metadata` see *Merging multiple structure-based fingerprint sources*.

Even though I used the fingerprint family to get the type, I did that more for pedagogical reasons. Most times you can get the fingerprint type directly using *chemfp.get_fingerprint_type()*. You can call it using a fingerprint type string or by passing in the parameters in the optional second parameter::

```
>>> fptype = chemfp.get_fingerprint_type("OpenEye-Path numbits=256")
>>> fptype = chemfp.get_fingerprint_type("OpenEye-Path", {"numbits": 256})
```

See *get_fingerprint_type() and get_type()* for examples on how to use `get_fingerprint_type`.

Fingerprint family

In this section you'll learn about the attributes and methods of a fingerprint family.

The *get_fingerprint_family()* function takes the fingerprint family name (with or without a version) and returns a *FingerprintFamily* instance:

```
>>> import chemfp
>>> family = chemfp.get_fingerprint_family("RDKit-Fingerprint")
```

It will raise a ValueError if you ask for a fingerprint family or version which doesn't exist:

```
>>> chemfp.get_fingerprint_family("whirl")
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "/Users/dalke/cvses/cfp-3x/docs/tmp/chemfp/__init__.py", line 1912, in get_
↪fingerprint_family
    return _family_registry.get_family(family_name)
  File "/Users/dalke/cvses/cfp-3x/docs/tmp/chemfp/types.py", line 1205, in get_family
    raise err
ValueError: Unknown fingerprint type 'whirl'
>>> chemfp.get_fingerprint_family("RDKit-Fingerprint/1")
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "/Users/dalke/cvses/cfp-3x/docs/tmp/chemfp/__init__.py", line 1912, in get_
↪fingerprint_family
    return _family_registry.get_family(family_name)
```

```
  File "/Users/dalke/cvses/cfp-3x/docs/tmp/chemfp/types.py", line 1205, in get_family
    raise err
ValueError: Unable to use RDKit-Fingerprint/1: This version of RDKit does not support␣
↪the RDKit-Fingerprint/1 fingerprint
```

The fingerprint family has several attributes to ask for the name or parts of the name:

```
>>> family
FingerprintFamily(<RDKit-Fingerprint/2>)
>>> family.name
'RDKit-Fingerprint/2'
>>> (family.base_name, family.version)
('RDKit-Fingerprint', '2')
```

It also has a `toolkit` attribute, which is the underlying chemfp toolkit that can create molecules for this fingerprint:

```
>>> family.toolkit
<module 'chemfp.rdkit_toolkit' from 'chemfp/rdkit_toolkit.pyc'>
>>> family.toolkit.name
'rdkit'
```

See the chapter *Toolkit API examples* for many examples of how to use a toolkit.

The *`get_defaults()`* method returns the default arguments used to create a fingerprint type, which is handy when you've forgotten what all of the arguments are:

```
>>> family.get_defaults()
{'maxPath': 7, 'fpSize': 2048, 'nBitsPerHash': 2, 'minPath': 1, 'useHs': 1}
```

If you call the family as a function, you'll get a *`FingerprintType`*. You can check to see that the fingerprint type's keyword arguments match the defaults:

```
>>> fptype = family()
>>> fptype.fingerprint_kwargs
{'maxPath': 7, 'fpSize': 2048, 'nBitsPerHash': 2, 'minPath': 1, 'useHs': 1}
```

Call the fingerprint family with keyword arguments to use something other than the default parameters:

```
>>> fptype = family(fpSize=1024, maxPath=6)
>>> fptype.fingerprint_kwargs
{'maxPath': 6, 'fpSize': 1024, 'nBitsPerHash': 2, 'minPath': 1, 'useHs': 1}
```

If you have the keyword arguments as a dictionary you can use the "**" syntax to apply the dictionary as keyword arguments, but I think it's clearer to call the *`FingerprintFamily.from_kwargs()`* method to create the fingerprint type:

```
>>> kwargs = {"fpSize": 512, "maxPath": 5}
>>> fptype = family(**kwargs) # Acceptable
>>> fptype.get_type()
'RDKit-Fingerprint/2 minPath=1 maxPath=5 fpSize=512 nBitsPerHash=2 useHs=1'
>>> fptype = family.from_kwargs(kwargs)  # Better
>>> fptype.get_type()
'RDKit-Fingerprint/2 minPath=1 maxPath=5 fpSize=512 nBitsPerHash=2 useHs=1'
```

(Currently `family(**kwargs)` forwards the the call to `family.from_kwargs(kwargs)` so there is a slight performance advantage to using `from_kwargs()`.)

Sometimes the fingerprint parameters come from a string, for example, from command-line arguments or a web form. In chemfp a dictionary of text keys and values are called "text settings". The fingerprint family has a helper function to process them and create a kwargs dictionary with the correct data types as values:

```
>>> family.get_kwargs_from_text_settings({
...     "fpSize": "128",
...     "nBitsPerHash": "1",
... })
{'maxPath': 7, 'fpSize': 128, 'nBitsPerHash': 1, 'minPath': 1, 'useHs': 1}
```

Note: This method is not as advanced as the *corresponding code in the toolkit Format API*. It does not understand namespaces. It will also raise an exception if called with an unsupported parameter:

```
>>> family.get_kwargs_from_text_settings({
...     "unsupported parameter": "-12.34",
... })
Traceback (most recent call last):
    ....
ValueError: Unsupported fingerprint parameter name 'unsupported parameter'
```

If you have text settings then you probably want to call *chemfp.get_fingerprint_type_from_text_settings()* directly instead of going through the fingerprint family:

```
>>> fptype = chemfp.get_fingerprint_type_from_text_settings("RDKit-Fingerprint",
...       {"fpSize": "512", "nBitsPerHash": "3", "maxPath": "6"})
>>> fptype.get_type()
'RDKit-Fingerprint/2 minPath=1 maxPath=6 fpSize=512 nBitsPerHash=3 useHs=1'
```

See *Create a fingerprint using text settings* for more examples of how to use this function.

Fingerprint family discovery

In this section you'll learn how to get the available fingerprint families, both as a set of name strings and a list of FingerprintFamily instances.

Even though chemfp knows about the OpenEye fingerprints, those fingerprints might not be available on your system if you don't have OEChem and OEGraphSim installed and licensed. Chemfp has a discovery system which will probe to see which fingerprint types are available and determine their version numbers.

If you just want the available family names, use *chemfp.get_fingerprint_family_names()*:

```
>>> import chemfp
>>> chemfp.get_fingerprint_family_names()
set(['RDKit-Torsion', 'OpenEye-Path', 'OpenBabel-FP2',
'OpenBabel-FP3', 'OpenBabel-FP4', 'RDKit-Avalon', 'RDMACCS-RDKit',
'RDKit-Morgan', 'OpenEye-MACCS166', 'RDMACCS-OpenEye',
'RDKit-MACCS166', 'OpenBabel-MACCS', 'ChemFP-Substruct-RDKit',
'ChemFP-Substruct-OpenEye', 'OpenEye-Circular', 'RDKit-Fingerprint',
'OpenEye-Tree', 'ChemFP-Substruct-OpenBabel', 'RDMACCS-OpenBabel',
'RDKit-AtomPair', 'RDKit-Pattern'])
```

Bear in mind that this might take a few seconds to run, since it will try to load the Python packages for each supported toolkit. (Once done, that list is cached so subsequent calls are fast.)

The function returns a set of base names, which don't contain the version information. Most likely you want to sort it before displaying it more nicely:

```
>>> from __future__ import print_function
>>> for name in sorted(chemfp.get_fingerprint_family_names()):
...   print(name)
...
ChemFP-Substruct-OpenBabel
ChemFP-Substruct-OpenEye
ChemFP-Substruct-RDKit
OpenBabel-FP2
OpenBabel-FP3
OpenBabel-FP4
OpenBabel-MACCS
OpenEye-Circular
OpenEye-MACCS166
OpenEye-Path
OpenEye-Tree
RDKit-AtomPair
RDKit-Avalon
RDKit-Fingerprint
RDKit-MACCS166
RDKit-Morgan
RDKit-Pattern
RDKit-Torsion
RDMACCS-OpenBabel
RDMACCS-OpenEye
RDMACCS-RDKit
```

On my desktop, where I do all of the testing, I have many virturalenv installations so I can test different combinations of Python and toolkit versions. I'll run chemfp in one of the OpenEye-only virtualenv installations and show that it only knows about the OEChem/OEGraphSim fingerprint types:

```
>>> from __future__ import print_function
>>> import chemfp
>>> print("\n".join(sorted(chemfp.get_fingerprint_family_names())))
ChemFP-Substruct-OpenEye
OpenEye-Circular
OpenEye-MACCS166
OpenEye-Path
OpenEye-Tree
RDMACCS-OpenEye
```

It's still possible to get a list of all fingerprint family names, including those which aren't actually available for the given Python installation, by setting the *include_unavailable* parameter to True:

```
>>> print("\n".join(sorted(chemfp.get_fingerprint_family_names(include_
↪unavailable=True))))
ChemFP-Substruct-OpenBabel
ChemFP-Substruct-OpenEye
ChemFP-Substruct-RDKit
OpenBabel-FP2
OpenBabel-FP3
OpenBabel-FP4
OpenBabel-MACCS
OpenEye-Circular
OpenEye-MACCS166
OpenEye-Path
OpenEye-Tree
RDKit-AtomPair
RDKit-Avalon
```

```
RDKit-Fingerprint
RDKit-MACCS166
RDKit-Morgan
RDKit-Pattern
RDKit-Torsion
RDMACCS-OpenBabel
RDMACCS-OpenEye
RDMACCS-RDKit
```

The list of base names is pretty useful, but sometimes you want more details, like the specific version number, and the default number of bits. The *FingerprintFamily* includes the attributes to get the *name* and *version* but it doesn't have a way to get the default number of bits. Instead, I'll use the FingerprintFamily to make a *FingerprintType* with the default parameters, then ask the new fingerprint type its *number of bits*.

This means I need a list of FingerprintFamily instances, which is conveniently available from *chemfp.get_fingerprint_families()*. (Remember, this may take a few seconds the first time it's called, because it tries to load all of the available fingerprints. Once determined, this information is cached.)

As a result, you can make a list of all available fingerprint methods and their default number of bits with the following:

```
>>> for family in chemfp.get_fingerprint_families():
...   print(family.name, family().num_bits)
...
ChemFP-Substruct-OpenBabel/1 881
ChemFP-Substruct-OpenEye/1 881
ChemFP-Substruct-RDKit/1 881
OpenBabel-FP2/1 1021
OpenBabel-FP3/1 55
OpenBabel-FP4/1 307
OpenBabel-MACCS/2 166
OpenEye-Circular/2 4096
OpenEye-MACCS166/3 166
OpenEye-Path/2 4096
OpenEye-Tree/2 4096
RDKit-AtomPair/2 2048
RDKit-Avalon/1 512
RDKit-Fingerprint/2 2048
RDKit-MACCS166/2 166
RDKit-Morgan/1 2048
RDKit-Pattern/2 2048
RDKit-Torsion/2 2048
RDMACCS-OpenBabel/2 166
RDMACCS-OpenEye/2 166
RDMACCS-RDKit/2 166
```

The output here is a bit fancy. If you only want the version information then you could just look at the list, since a family's repr shows the versioned name:

```
>>> chemfp.get_fingerprint_families()
[FingerprintFamily(<ChemFP-Substruct-OpenBabel/1>), FingerprintFamily(<ChemFP-
↪Substruct-OpenEye/1>),
FingerprintFamily(<ChemFP-Substruct-RDKit/1>), FingerprintFamily(<OpenBabel-FP2/1>),
FingerprintFamily(<OpenBabel-FP3/1>), FingerprintFamily(<OpenBabel-FP4/1>),
FingerprintFamily(<OpenBabel-MACCS/2>), FingerprintFamily(<OpenEye-Circular/2>),
FingerprintFamily(<OpenEye-MACCS166/3>), FingerprintFamily(<OpenEye-Path/2>),
FingerprintFamily(<OpenEye-Tree/2>), FingerprintFamily(<RDKit-AtomPair/2>),
FingerprintFamily(<RDKit-Avalon/1>), FingerprintFamily(<RDKit-Fingerprint/2>),
FingerprintFamily(<RDKit-MACCS166/2>), FingerprintFamily(<RDKit-Morgan/1>),
```

```
FingerprintFamily(<RDKit-Pattern/2>),  FingerprintFamily(<RDKit-Torsion/2>),
FingerprintFamily(<RDMACCS-OpenBabel/2>),  FingerprintFamily(<RDMACCS-OpenEye/2>),
FingerprintFamily(<RDMACCS-RDKit/2>)]
```

On the other hand, that's a rather dense block of text.

Finally, use *chemfp.has_fingerprint_family()* to test if a fingerprint family is available:

```
>>> chemfp.has_fingerprint_family("OpenEye-Tree")
True
>>> chemfp.has_fingerprint_family("OpenEye-Tree/2")
True
>>> chemfp.has_fingerprint_family("OpenEye-Tree/1")
False
```

It understands both version and unversioned names.

get_fingerprint_type() and get_type()

In this section you'll learn how to get a fingerprint type given its type string, and how to specify fingerprint parameters as a dictionary.

The easiest way to get a specific *FingerprintType* is with *chemfp.get_fingerprint_type()*:

```
>>> import chemfp
>>> fptype = chemfp.get_fingerprint_type("RDKit-Fingerprint")
>>> fptype
<chemfp.rdkit_types.RDKitFingerprintType_v2 object at 0x10cfedb10>
```

The fingerprint type has a *FingerprintType.get_type()* method, which returns the canonical fingerprint type string:

```
>>> fptype.get_type()
'RDKit-Fingerprint/2 minPath=1 maxPath=7 fpSize=2048 nBitsPerHash=2 useHs=1'
```

This is canonical because chemfp ensures that all fingerprint type strings with the same parameter values have the same type string.

I left out the version number in the fingerprint name, so chemfp gives me the most recent supported version. I could have included the version in the name, which is useful if you want to prevent a version mismatch between your data sets. If the version doesn't exist, the function will raise a ValueError:

```
>>> fptype = chemfp.get_fingerprint_type("RDKit-Fingerprint/2")
>>> chemfp.get_fingerprint_type("RDKit-Fingerprint/1")
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "chemfp/__init__.py", line 1984, in get_fingerprint_type
    return types.registry.get_fingerprint_type(type, fingerprint_kwargs)
  File "chemfp/types.py", line 1233, in get_fingerprint_type
    raise ValueError("Unable to use %s: %s" % (name, reason))
ValueError: Unable to use RDKit-Fingerprint/1: This version of
RDKit does not support the RDKit-Fingerprint/1 fingerprint
```

I can also specify some or all of the parameters myself in the type string, instead of accepting the default values:

```
>>> fptype = chemfp.get_fingerprint_type("RDKit-Fingerprint fpSize=1024 maxPath=6")
>>> fptype.get_type()
'RDKit-Fingerprint/2 minPath=1 maxPath=6 fpSize=1024 nBitsPerHash=2 useHs=1'
```

You can also pass in the parameters as a Python dictionary, though you still need at least the base name of the fingerprint family:

```
>>> fp_kwargs = {
...   "maxPath": 6,
...   "fpSize": 512,
... }
>>> fptype = chemfp.get_fingerprint_type("RDKit-Fingerprint", fp_kwargs)
>>> fptype.get_type()
'RDKit-Fingerprint/2 minPath=1 maxPath=6 fpSize=512 nBitsPerHash=2 useHs=1'
```

If a parameter is specified in both the type string and the dictionary then the dictionary value will be used:

```
>>> fptype = chemfp.get_fingerprint_type("RDKit-Fingerprint fpSize=1024 minPath=2",
...                                      {"fpSize": 128})
>>> fptype.get_type()
'RDKit-Fingerprint/2 minPath=2 maxPath=7 fpSize=128 nBitsPerHash=2 useHs=1'
```

Create a fingerprint using text settings

In this section you'll learn how to get a fingerprint type using text settings.

The fingerprint keywords arguments ("kwargs") are a dictionary whose keys are fingerprint parameter names and whose values are native Python objects for those parameters. Here is a fingerprint kwargs dictionary for the RDKit-Fingerprint:

```
{'maxPath': 7, 'fpSize': 2048, 'nBitsPerHash': 2, 'minPath': 1, 'useHs': 1}
```

Text settings are a dictionary where the dictionary keys are still parameter names but where the dictionary values are string-encoded parameter values. Here is the equivalent text settings for the above kwargs dictionary:

```
{'maxPath': '7', 'fpSize': '2048', 'nBitsPerHash': '2', 'minPath': '1', 'useHs': '1'}
```

A text settings dictionary typically comes from command-line parameters or a configuration file, where everything is a string. The fingerprint family has a method to convert text settings to kwargs:

```
>>> import chemfp
>>> family = chemfp.get_fingerprint_family("RDKit-Fingerprint")
>>> kwargs = family.get_kwargs_from_text_settings({"fpSize": "4096"})
>>> kwargs
{'maxPath': 7, 'fpSize': 4096, 'nBitsPerHash': 2, 'minPath': 1, 'useHs': 1}
```

The kwargs can then be used to get the specified fingerprint type from the family:

```
>>> fptype = family.from_kwargs(kwargs)
>>> fptype
<chemfp.rdkit_types.RDKitFingerprintType_v2 object at 0x100f68610>
>>> fptype.get_type()
'RDKit-Fingerprint/2 minPath=1 maxPath=7 fpSize=4096 nBitsPerHash=2 useHs=1'
```

It's a bit tedious to go through all those steps to process some text settings. Instead, call *chemfp.get_fingerprint_type_from_text_settings()*:

```
>>> fptype = chemfp.get_fingerprint_type_from_text_settings(
...                     "RDKit-Fingerprint", {"fpSize": "4096"})
>>> fptype.get_type()
'RDKit-Fingerprint/2 minPath=1 maxPath=7 fpSize=4096 nBitsPerHash=2 useHs=1'
```

The parameters in the text settings have priority should the fingerprint type string and the text settings both specify the same parameter name, as in this example where the fingerprint type string specifies a 1024 bit fingerprint while the text settings specifies a 4096 bit fingerprint:

```
>>> fptype = chemfp.get_fingerprint_type_from_text_settings("RDKit-Fingerprint␣
↪fpSize=1024")
>>> fptype.get_type()
'RDKit-Fingerprint/2 minPath=1 maxPath=7 fpSize=1024 nBitsPerHash=2 useHs=1'
>>>
>>> fptype = chemfp.get_fingerprint_type_from_text_settings(
...             "RDKit-Fingerprint fpSize=1024", {"fpSize": "4096"})
>>> fptype.get_type()
'RDKit-Fingerprint/2 minPath=1 maxPath=7 fpSize=4096 nBitsPerHash=2 useHs=1'
```

At present there is no support for parameter namespaces, and unknown parameter names will raise an exception:

```
>>> fptype = chemfp.get_fingerprint_type_from_text_settings(
...             "RDKit-Fingerprint", {"fpSize": "4096", "spam": "eggs"})
Traceback (most recent call last):
  File "<stdin>", line 2, in <module>
  File "chemfp/__init__.py", line 1329, in get_fingerprint_type_from_text_settings
    return types.registry.get_fingerprint_type_from_text_settings(type, settings)
  File "chemfp/types.py", line 868, in get_fingerprint_type_from_text_settings
    raise ValueError("Error with type %r: %s" % (type, err))
ValueError: Error with type 'RDKit-Fingerprint': Unsupported fingerprint parameter␣
↪name 'spam'
```

This may change in the future; let me know what's best for you.

For now, if you want to remove unexpected names from a dictionary then use the fingerprint family's *get_defaults()* to get the default kwargs as a dictionary, and use the keys to filter out the unknown parameters:

```
>>> family = chemfp.get_fingerprint_family("RDKit-Fingerprint")
>>> defaults = family.get_defaults()
>>> defaults
{'maxPath': 7, 'fpSize': 2048, 'nBitsPerHash': 2, 'minPath': 1, 'useHs': 1}
>>> settings = {"maxPath": "8", "unknown": "mystery"}
>>> new_settings = dict((k, v) for (k,v) in settings.items() if k in defaults)
>>> new_settings
{'maxPath': '8'}
```

FingerprintType properties and methods

In this section you'll learn about the *FingerprintType* properties and methods.

I'll start by getting OpenEye's tree fingerprint using the default parameters:

```
>>> fptype = chemfp.get_fingerprint_type("OpenEye-Tree")
>>> fptype
<chemfp.openeye_types.OpenEyeTreeFingerprintType_v2 object at 0x10a64be10>
```

```
>>> fptype.get_type()
'OpenEye-Tree/2 numbits=4096 minbonds=0 maxbonds=4␣
↪atype=Arom|AtmNum|Chiral|FCharge|HvyDeg|Hyb btype=Order'
```

The "OpenEye-Tree/2" is the fingerprint *name*, which is decomposed into the *base_name* "OpenEye-Tree" and the *version* "2":

```
>>> fptype.name
'OpenEye-Tree/2'
>>> fptype.base_name, fptype.version
('OpenEye-Tree', '2')
```

The number of bits for the fingerprint is *num_bits*, and *fingerprint_kwargs* is a fingerprint parameters as a dictionary of Python values:

```
>>> fptype.num_bits
4096
>>> fptype.fingerprint_kwargs
{'maxbonds': 4, 'numbits': 4096, 'atype': 63, 'minbonds': 0, 'btype': 1}
```

Each fingerprint type has a *toolkit*, which is the chemfp toolkit that can make molecules used as input to the fingerprint type. (This would be None if there were no toolkit.) Given a fingerprint type it's easy to figure out the *toolkit.name* of the toolkit it's associated with:

```
>>> fptype.toolkit.name
'openeye'
```

The *software* attribute gives information about the software used to generate the fingerprint. For RDKit and Open Babel this is the same as the *toolkit.software* string. On the other hand, OpenEye distributes OEChem and OEGraphSim as two different libraries. These map quite naturally to chemfp's concepts of fingerprint type and toolkit, so the "software" field for its fingerprint type and toolkit differ:

```
>>> fptype.software
'OEGraphSim/2.2.6 (20170208) chemfp/3.1'
>>> fptype.toolkit.software
'OEChem/20170208'
```

Finally, *FingerprintType.get_fingerprint_family()* returns the fingerprint family for a given fingerprint type:

```
>>> fptype.get_fingerprint_family()
FingerprintFamily(<OpenEye-Tree/2>)
```

Convert a structure record to a fingerprint

In this section you'll learn how to use a fingerprint type to convert a structure record into a fingerprint.

The *FingerprintType* method *parse_molecule_fingerprint()* parses a structure record and returns the fingerprint as a byte string. The following uses Open Babel to get the MACCS fingerprint for phenol:

```
>>> import chemfp
>>> from chemfp import bitops
>>> fptype = chemfp.get_fingerprint_type("OpenBabel-MACCS")
>>> fptype
<chemfp.openbabel_types.OpenBabelMACCSFingerprintType_v2 object at 0x10cfedc10>
>>> fp = fptype.parse_molecule_fingerprint("c1ccccc1O", "smistring")
```

```
>>> fp
'\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x01@\x00D\x80\x10\x1e'
>>> bitops.hex_encode(fp)
'00000000000000000000000000000140004480101e'
```

(Under Python 3 the fingerprint is a byte string and the second-to-last output line will be shown with the b" prefix.)

The parameters to `parse_molecule_fingerprint()` are identical to the toolkit's *`parse_molecule()`* function. For example, the following shows that the SMILES "Q" raises a *`chemfp.ParseError`* with the default *errors* mode, and returns None when *errors* is "ignore":

```
>>> fptype.parse_molecule_fingerprint("Q", "smistring")
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "chemfp/types.py", line 984, in parse_molecule_fingerprint
    mol = self.toolkit.parse_molecule(content, format, reader_args=reader_args,␣
↪errors=errors)
          .... lines omitted ...
  File "chemfp/io.py", line 87, in error
    _compat.raise_tb(ParseError(msg, location), None)
  File "<string>", line 1, in raise_tb
chemfp.ParseError: Open Babel cannot parse the SMILES 'Q'
>>> fptype.parse_molecule_fingerprint("Q", "smistring", errors="ignore") is None
True
```

See *Parse and create SMILES* for information about using `parse_molecule()` and the distinction between "smistring", "smi" and other SMILES formats. See *Specify alternate error behavior* for more about the *errors* parameter.

Convert a structure record to an id and fingerprint

In this section you'll learn how to use a fingerprint type to extract the id from a structure record, convert the structure record into a fingerprint, and return the (id, fingerprint) pair.

The previous section showed how to convert a structure record into a fingerprint. Sometimes you'll also want the identifier. The *`FingerprintType`* method *`parse_id_and_molecule_fingerprint()`* does both in the same call.

```
>>> fptype = chemfp.get_fingerprint_type("OpenEye-MACCS166")
>>> fptype.parse_id_and_molecule_fingerprint("c1ccccc1O phenol", "smi")
(u'phenol',
↪'\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x01@\x00\x04\x00\x10\x1a')
```

(If the identifier is not present then the function may return None or the empty string, depending on the format and underlying implementation.)

The parameters to `parse_id_and_molecule_fingerprint` are identical to the *`toolkit.parse_id_and_molecule()`* function. For example, the following shows the difference in using two different delimiter types in the *reader_args*:

```
>>> record = "C1C(C)=C(C=CC(C)=CC=CC(C)=CCO)C(C)(C)C1 vitamin a"
>>> fptype.parse_id_and_molecule_fingerprint(record, "smi", reader_args={"delimiter":
↪"to-eol"})
(u'vitamin a',
↪'\x00\x00\x00\x08\x00\x00\x02\x00\x02\n\x02\x80\x04\x98\x0c\x00\x00\x140\x14\x18')
>>> fptype.parse_id_and_molecule_fingerprint(record, "smi", reader_args={"delimiter":
↪"space"})
```

```
(u'vitamin',
↪'\x00\x00\x00\x08\x00\x00\x02\x00\x02\n\x02\x80\x04\x98\x0c\x00\x00\x140\x14\x18')
```

The *id_tag* and *errors* parameters are also supported, though I won't give examples. See *Read ids and molecules using an SD tag for the id* to learn how to use the *id_tag* and *Specify a SMILES delimiter through reader_args* and *Multi-toolkit reader_args and writer_args* for examples of using *reader_args*.

Make a specialized id and molecule fingerprint parser

In this section you'll learn how to make a specialized function for computing the fingerprints given many individual structure records.

Sometimes the structure input comes as a set of individual strings, with one record per string. For example, the input might come from a database query, where the cursor returns each field of each row as its own term, and you want to convert each of them into a fingerprint.

One way to do this through successive calls to *FingerprintType.parse_molecule_fingerprint()*:

```
>>> from __future__ import print_function
>>> import chemfp
>>> from chemfp import bitops
>>>
>>> smiles_list = ["C", "O=O", "C#N"]
>>>
>>> fptype = chemfp.get_fingerprint_type("RDKit-MACCS166")
>>> for smiles in smiles_list:
...     fp = fptype.parse_molecule_fingerprint(smiles, "smistring")
...     print(bitops.hex_encode(fp), smiles)
...
00000000000000000000000000000000000008000 C
00000000000000000000000200000080000004008 O=O
00000000001000000000000000000000000000001 C#N
```

There is some overhead in this because the parameters, like *format* ("smistring" in this case) are (re)validated for each call, and sometimes extra work is done to ensure that the call is thread-safe. (The overhead is higher if there are complex reader args, and if the underlying fingerprinter is very fast.)

Another solution is to use `make_id_and_molecule_fingerprinter_parser()` to create a specialized parser function for a given set of parameters. The parameters are only validated once, and the returned parser function takes only the record as input and returns the (id, fingerprint) pair:

```
>>> import chemfp
>>> fptype = chemfp.get_fingerprint_type("RDKit-MACCS166")
>>> id_and_fp_parser = fptype.make_id_and_molecule_fingerprint_parser("smi")
>>> id_and_fp_parser("c1ccccc1O phenol")
(u'phenol',
↪'\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x01@\x00D\x80\x10\x1e')
```

The parameters to `make_id_and_molecule_fingerprint_parser` are identical to *toolkit.make_id_and_molecule_parser()*.

I'll use the new function to parse the `smiles_list` from earlier:

```
>>> from __future__ import print_function
>>> import chemfp
>>> from chemfp import bitops
>>>
```

```
>>> smiles_list = ["C", "O=O", "C#N"]
>>>
>>> fptype = chemfp.get_fingerprint_type("RDKit-MACCS166")
>>> id_and_fp_parser = fptype.make_id_and_molecule_fingerprint_parser("smistring")
>>>
>>> for smiles in smiles_list:
...     id, fp = id_and_fp_parser(smiles)
...     print(bitops.hex_encode(fp), smiles)
...
00000000000000000000000000000000000008000 C
00000000000000000000000020000080000004008 O=O
00000000001000000000000000000000000000001 C#N
```

For OpenEye-MACCS166, creating and using a specialized parser is about 15% faster than using the parse_molecule_fingerprint() when the query is isocane (C20H42). For OpenBabel-MACCS it's about 5%, and for RDKit-MACCS166 it's around 1%.

You might be tempted to assume there's a constant Python overhead and use the above numbers to judge the performance of the underlying toolkit. This won't give accurate answers. Chemfp makes certain threading guarantees which aren't always directly mapped to the underlying toolkit. This can require extra overhead.

In addition, RDKit's native MACCS implementation maps key 1 to bit 1, while the other toolkits and chemfp map key 1 to bit 0. Chemfp normalizes RDKit-MACCS by shifting all of the bits left, and this translation code hasn't yet been optimized.

You may have noticed that there's a `parse_molecule_fingerprint()` and a `make_id_and_molecule_fingerprint_parser()` but there isn't a `parse_id_and_molecule_fingerprint()` or `make_molecule_fingerprint_parser()`. This is simply a matter of time. I haven't needed those functions, they are quite easy to emulate given what's available, and I was getting bored of writing test cases.

Let me know if they would be useful for your code.

Read a structure file and compute fingerprints

In this section you'll learn how to use a fingerprint type to read a structure file, compute fingerprints for each one, and iterate over the resulting (id, fingerprint) pairs. You will need Compound_027575001_027600000.sdf.gz from PubChem.

The *read_molecule_fingerprints()* method of a *FingerprintType* reads a structure file and computes the fingerprint for each molecule. It will also extract the record identifier. It returns an iterator of the (id, fingerprint) pairs. For example, the following uses OEChem/OEGraphSim to compute the MACCS166 fingerprint for a PubChem file, and prints the identifier, the number of keys set in the fingerprint, and the hex-encoded fingerprint:

```
from __future__ import print_function
import chemfp
from chemfp import bitops

# Uncomment the fingerprint type you want to use.
fptype = chemfp.get_fingerprint_type("OpenEye-MACCS166")
#fptype = chemfp.get_fingerprint_type("RDKit-MACCS166")
#fptype = chemfp.get_fingerprint_type("OpenBabel-MACCS")
for id, fp in fptype.read_molecule_fingerprints("Compound_014550001_014575000.sdf.gz
↪"):
    print("%s %3d %s" % (id, bitops.byte_popcount(fp), bitops.hex_encode(fp)))
```

The first few lines of ouput (excluding OEChem warnings) are:

```
14550001  46 00008008000081406000e226a0906142df8e2a7c1b
14550002  35 00000008000000000000ea06809021425d862a7c1b
14550003  25 00000008000000000000c812008005425084283c1b
14550004  23 00000000000000800118204a00080800300b0780813
14550005  16 00000000040001000000000010010004800803523a
```

However, in most cases you should use the top-level helper function *chemfp.read_molecule_fingerprints()*, which does the fingerprint type lookup and the call to read_molecule_fingerprints:

```
from __future__ import print_function
import chemfp
from chemfp import bitops

for id, fp in chemfp.read_molecule_fingerprints("OpenEye-MACCS166",
                                                "Compound_014550001_014575000.sdf.gz
↪"):
    print("%s %3d %s" % (id, bitops.byte_popcount(fp), bitops.hex_encode(fp)))
```

The helper function accepts both a type string, as shown here, and a Metadata object. On the other hand, the helper function does not support fingerprint kwargs, so in that case you have to go through the FingerprintType.

The read_molecule_fingerprints method takes the same parameters as the toolkit.read_ids_and_molecules(), including *id_tag*, *errors*, and *location*. I won't cover those details again here. Instead, see *Read ids and molecules from an SD file at the same time*.

Structure-based fingerprint reader location

In this section you'll learn more about the location attribute of the structure-based fingerprint iterator returned by read_molecule_fingerprints and read_molecule_fingerprints_from_string.

Four related functions implement structure-based fingerprint readers:

- *chemfp.read_molecule_fingerprints()*
- *chemfp.read_molecule_fingerprints_from_string()*
- *FingerprintType.read_molecule_fingerprints()*
- *FingerprintType.read_molecule_fingerprints_from_string()*

They all return a *FingerprintIterator*. Just like with the *BaseMoleculeReader* classes, the FingerprintIterator has a *location* attribute that can be used to get more information about the internal reader state. The toolkit section has more details about how to get the current record number (see *Location information: filename, record_format, recno and output_recno*) and, if supported by the parser implementation for a format, the line number and byte ranges for the record (see *Location information: record position and content*).

It's also possible to get the current molecule object using the location's "mol" attribute. This isn't so important for the toolkit API since all of the molecule readers return the molecule object. It's more useful in the fingerprint iterator, which doesn't.

NOTE: accessing the molecule this way is somewhat slow, because it requires several Python function calls. It should mostly be used for error reporting; the following is meant as an example of use, and not a recommended best practice.

The following uses the location's mol to report the SMILES string for every molecule whose MACCS fingerprint sets fewer than 5 keys:

```
from __future__ import print_function
import chemfp
from chemfp import bitops

from openeye.oechem import OECreateSmiString

fptype = chemfp.get_fingerprint_type("OpenEye-MACCS166")
with fptype.read_molecule_fingerprints("Compound_014550001_014575000.sdf.gz") as␣
↪reader:
    location = reader.location
    for id, fp in reader:
        popcount = bitops.byte_popcount(fp)
        if popcount >= 5:
            continue
        smiles = OECreateSmiString(location.mol)
        print("%s %3d %s" % (id, popcount, smiles))
```

The output from the above is:

```
14550474   3 [Mg+2].[Ca+2]
14567810   4 [B]=CO
14574228   4 F[In]
14574262   3 [Ga].[Ga].[Ga].[Ga].[Ga].[Ir].[Ir].[Ir]
14574264   3 [Co].[Ga]
14574265   3 [Ga].[Ga].[Pt]
14574267   3 [Ga].[Pt]
14574635   4 [Mg+2].[Al+3]
14574653   4 [Na+].[Na+].[Na+].[PH2-]
```

The above code imports and calls OECreateSmiString directly. The cross-toolkit solution is only slightly more complicated. I need to use the fingerprint type object to get the underlying "toolkit", which is a portability layer on top of the actual cheminformatics toolkit with functions to parse a string into a molecule and vice versa:

```
>>> import chemfp
>>> fptype = chemfp.get_fingerprint_type("OpenEye-MACCS166")
>>> fptype.toolkit
<module 'chemfp.openeye_toolkit' from '/Users/dalke/cvses/cfp-3x/docs/tmp/chemfp/
↪openeye_toolkit.py'>
>>> T = fptype.toolkit
>>> mol = T.parse_molecule("OC", "smistring")
>>> T.create_string(mol, "smistring")
'CO'
```

I'll use the toolkit's `create_string()` method to make the SMILES string for each molecule which passes the filter:

```
from __future__ import print_function
import chemfp
from chemfp import bitops

fptype = chemfp.get_fingerprint_type("OpenEye-MACCS166")
T = fptype.toolkit

with fptype.read_molecule_fingerprints("Compound_014550001_014575000.sdf.gz") as␣
↪reader:
    location = reader.location
    for id, fp in reader:
        popcount = bitops.byte_popcount(fp)
```

```
        if popcount >= 5:
            continue
        smiles = T.create_string(location.mol, "smistring")
        print("%s %3d %s" % (id, popcount, smiles))
```

When should you use a toolkit-specific API and when to use the portable one?

That depends on you. There's definitely a portability vs. performance tradeoff because the new `create_string` function will always require an extra function call over the native API. If you work with a given toolkit a lot then you're going to be more familiar with it than this brand new chemfp API. Plus, calling a function to create another function is somewhat unusual.

On the other hand, it's trivial to change the above code to work with any of the fingerprint types that chemfp supports.

Read fingerprints from a string containing structures

In this section you'll learn how to use a fingerprint type to read a string containing a set of structure records, compute fingerprints for each one, and iterate over the resulting (id, fingerprint) pairs.

The *read_molecule_fingerprints_from_string()* method of the *FingerprintType* takes as input a string containing structure records and returns an iterator over the (id, fingerprint) pairs.

```
>>> from __future__ import print_function
>>> import chemfp
>>> from chemfp import bitops
>>> fptype = chemfp.get_fingerprint_type("OpenBabel-MACCS")
>>> content = "C methane\n" + "CC ethane\n"
>>> reader = fptype.read_molecule_fingerprints_from_string(content, "smi")
>>> for (id, fp) in reader:
...   print(id, bitops.hex_encode(fp))
...
methane 000000000000000000000000000000000008000
ethane 000000000000000000000000000000000108000
>>>
```

In most cases you should use the top-level helper function *chemfp.read_molecule_fingerprints_from_string()*, which is slightly easier to call:

```
from __future__ import print_function
import chemfp
from chemfp import bitops
content = ("C methane\n"
           "CC ethane\n")
reader = chemfp.read_molecule_fingerprints_from_string("OpenBabel-MACCS",
                                                       content, "smi")
for (id, fp) in reader:
    print(id, bitops.hex_encode(fp))
```

The helper function accepts both a type string, as shown here, and a `Metadata` object. The helper function does not support fingerprint kwargs so in that case you must go through the fingerprint type.

The method takes the same parameters as `toolkit.read_ids_and_molecules_from_string()`, including the *id_tag*, *errors*, *location*, and *reader_args*. See *Read from a string instead of a file* for more about that function.

Structure-based fingerprint reader errors

In this section you'll learn how to use the *errors* option for the "read molecule fingerprints" functions, including how to use the experimental support for a callback error handler.

The four structure reader functions (*chemfp.read_molecule_fingerprints()*, *chemfp.read_molecule_fingerprints_from_string()*, *FingerprintType.read_molecule_fingerprints()*, and *FingerprintType.read_molecule_fingerprints_from_string()*) take the standard *errors* option. By default it is "strict", which means that it raises an exception when there are errors, and stops processing.

```
>>> from __future__ import print_function
>>> import chemfp
>>> from chemfp import bitops
>>> content = ("C methane\n" +
...            "Q Q-ane\n" +
...            "O=O molecular oxygen\n")
>>> with chemfp.read_molecule_fingerprints_from_string(
...           "RDKit-MACCS166", content, "smi") as reader:
...   for (id, fp) in reader:
...     print(id, bitops.hex_encode(fp))
...
methane 00000000000000000000000000000000000008000
[02:19:12] SMILES Parse Error: syntax error for input: 'Q'
Traceback (most recent call last):
        ....
  File "chemfp/io.py", line 87, in error
    _compat.raise_tb(ParseError(msg, location), None)
  File "<string>", line 1, in raise_tb
chemfp.ParseError: RDKit cannot parse the SMILES 'Q', file '<string>', line 2, record
↪#2: first line is 'Q Q-ane'
```

The default is "strict" because you should be the one to decide if you really want to ignore errors, not me. Specify `errors="ignore"` to ignore errors, or use "report" to have chemfp write its own error messages to stderr:

```
>>> with chemfp.read_molecule_fingerprints_from_string(
...           "RDKit-MACCS166", content, "smi", errors="ignore") as reader:
...   for (id, fp) in reader:
...     print(id, bitops.hex_encode(fp))
...
methane 00000000000000000000000000000000000008000
[02:21:36] SMILES Parse Error: syntax error for input: 'Q'
molecular oxygen 00000000000000000000000020000080000004008
```

Of course, this depends on the underlying toolkit implementation. Some toolkit/format combinations don't let chemfp know there was an error, such as most of the OEChem-based formats.

Experimental error handler

In this section you'll learn about the experimental API for writing your own error handler.

In the previous section you learned about the "strict", "report", and "ignore" error handlers. What if you want something different? Chemfp has an experimental feature where the *errors* can be any object with the method "error(message, location)". You might send the results to a log file, or display it in a GUI, ... or send it to a speech synthesizer and hear all of the error messages go by.

NOTE: This error handler API is experimental and may change in the future.

The following creates an error handler which counts the number of errors, and for each one reports the error number, the filename (which is "<string>" if the input is from a string), and the error message:

```
>>> class ErrorCounter(object):
...     def __init__(self):
...         self.num_errors = 0
...     def error(self, message, location):
...         self.num_errors += 1
...         print("Failure #%d from file %r: %s" % (
...                self.num_errors, location.filename, message))
...
>>> error_handler = ErrorCounter()
>>> # ... use  'content' from the previous section
>>> with chemfp.read_molecule_fingerprints_from_string(
...           "RDKit-MACCS166", content, "smi", errors=error_handler) as reader:
...     for (id, fp) in reader:
...         print(id, bitops.hex_encode(fp))
...
methane 00000000000000000000000000000000000008000
[02:26:02] SMILES Parse Error: syntax error for input: 'Q'
Failure #1 from file '<string>': RDKit cannot parse the SMILES 'Q'
molecular oxygen 00000000000000000000000200000080000004008
```

Let me know if you use the API and have ideas for improvements.

The *toolkit documentation* includes another example of how to write an error handler.

Compute a fingerprint for a native toolkit molecule

In this section you'll learn how to compute a fingerprint given a toolkit molecule.

All of the previous sections assumed the inputs were structure record(s), either as a string or from a file. What if you already have a native toolkit molecule and want to compute its fingerprint? In that case, use the *FingerprintType.compute_fingerprint()* method:

```
>>> import chemfp
>>> fptype = chemfp.get_fingerprint_type("OpenBabel-MACCS")
>>> mol = fptype.toolkit.parse_molecule("c1ccccc1O", "smistring")
>>> mol
<openbabel.OBMol; proxy of <Swig Object of type 'OpenBabel::OBMol *' at 0x10d2bf510> >
>>> fptype.compute_fingerprint(mol)
'\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x01@\x00D\x80\x10\x1e'
```

This can be useful when you want to compute multiple fingerprint types for the same molecule. For example, I'll compare Open Babel's MACCS implementation with chemfp's own MACCS implementation for Open Babel:

```
from __future__ import print_function
import chemfp
from chemfp import openbabel_toolkit as T
from chemfp import bitops

fptype1 = chemfp.get_fingerprint_type("OpenBabel-MACCS")
fptype2 = chemfp.get_fingerprint_type("RDMACCS-OpenBabel")

with T.read_ids_and_molecules("Compound_014550001_014575000.sdf.gz") as reader:
    for id, mol in reader:
        fp1 = fptype1.compute_fingerprint(mol)
        fp2 = fptype2.compute_fingerprint(mol)
```

```
        if fp1 != fp2:
            bits1 = set(bitops.byte_to_bitlist(fp1))
            bits2 = set(bitops.byte_to_bitlist(fp2))
            print(id, "in OB:", sorted(bits1-bits2), "in RDMACCS:", sorted(bits2-
↪bits1))
        else:
            print(id, "equal")
```

Almost half (2186 of 5167) of the output were lines of the form:

```
14574962 in OB: [] in RDMACCS: [124]
```

I was curious, so I investigated the differences. Key 125 (the MACCS keys start at 1 while chemfp bit indexing starts at 0) is defined as "Aromatic Ring > 1". Open Babel doesn't support this bit because it only allows key definitions based on SMARTS, and this query cannot be represented as SMARTS.

This is also why there are 90 records where chemfp's RDMACCS finds bit 165/key 166 ("more than one fragment"). That can be expressed as the SMARTS "(*).(*)" but when the MACCS definitions were added to Open Babel it didn't understand component level groupings, so that pattern was omitted, and Open Babel will always generate a 0 for it. Always, that is, until someone implements it. (Might that be you?)

For the record, 2756 of the records matched exactly, 2186 set bit 124 in RDMACCS, 90 set bit 165 in RDMACCS, 123 set both bit 124 and 165 in RDMACCS, and 1 set bit 111 in Open Babel's MACCS but *not* in RDMACCS while setting bit 124 in RDMACCS but *not* in Open Babel. I haven't investigated when PubChem record 14559073 gives this difference.

Note: `compute_fingerprint()` is thread-safe. If an underlying chemistry toolkit object is not thread-safe then chemfp will duplicate that object before computing the fingerprint.

Fingerprint many native toolkit molecules

In this section you'll learn how to generate a fingerprint given many native toolkit molecules.

Sometimes you have a list of molecules and you want to compute fingerprints for each one. In the following I'll load 4378 molecules from an SD file using OEChem:

```
 >>> import chemfp
 >>>
 >>> fptype = chemfp.get_fingerprint_type("OpenEye-MACCS166")
 >>> T = fptype.toolkit
 >>>
 >>> with T.read_molecules("Compound_014550001_014575000.sdf.gz") as reader:
 ...     mols = [T.copy_molecule(mol) for mol in reader]
 ...
              ... various OEChem warnings omitted ...
>>> len(mols)
5167
```

NOTE: for performance reasons, some of the toolkit implementations will reuse a molecule object. I call *toolkit.copy_molecule()* to force a copy of each one. A future version of chemfp will likely support a new *reader_args* parameter to ask the reader implementation to always return a new molecule.

You know from the previous section how to compute the fingerprint one molecule at a time using `FingerprintType.compute_fingerprint()`:

```
>>> fps = [fptype.compute_fingerprint(mol) for mol in mols]
```

You can also process all of them at once using `FingerprintType.compute_fingerprints()`:

```
>>> fps = list(fptype.compute_fingerprints(mols))
```

The plural in the name `compute_fingerprints()` is the hint that it can take multiple molecules. It returns a generator, so I used Python's `list()` to convert it to an actual list.

Why call `compute_fingerprints` instead of `compute_fingerprint`? The main reason is that it expresses your intent more clearly than setting up a for-loop. But to be honest, the original reason was that I expected it would be faster than calling the `compute_fingerprint` many times, because the underlying code could skip some overhead.

By design, `compute_fingerprint` is thread-safe, which means chemfp sometimes makes extra objects to keep that promise. On the other hand, `compute_fingerprints`, which processes a sequential series of molecules, can reuse internal objects across the series instead of creating new ones. In principle this should be a bit faster. In practice, nearly all of the time is spent in generating the fingerprint. Even with a faster fingerprint like OpenEye-Path, the timing difference is well under 1%, and not enough to be interesting.

Make a specialized molecule fingerprinter

In this section you'll learn how to make a specialized function to compute a fingerprint for a molecule. However, there is very little reason for you to use this function.

The `FingerprintType.compute_fingerprint()` method is thread-safe. Some of the underlying toolkit implementations can use code which isn't thread-safe. For example, OEGraphSim writes its fingerprint information to an OEFingerPrint instance, and replaces its previous value. A thread-safe implementation would make a new OEFingerPrint for each call, which a non-thread-safe implementation could reuse it, and save a small bit of allocation overhead.

The `FingerprintType.make_fingerprinter()` method returns a non-thread-safe fingerprinter function, which is potentially faster beause it doesn't need to keep the thread-safe promise.

Here's an example of the two APIs. First, a bit of preamble to get things set up with a couple of molecules:

```
>>> import chemfp
>>> from chemfp import bitops
>>>
>>> fptype = chemfp.get_fingerprint_type("OpenBabel-FP2")
>>> mol1 = fptype.toolkit.parse_molecule("c1ccccc1O", "smistring")
>>> mol2 = fptype.toolkit.parse_molecule("O=O", "smistring")
```

The thread-safe API calls the `compute_fingerprint()` method:

```
>>> bitops.byte_popcount(fptype.compute_fingerprint(mol1))
12
>>> bitops.byte_popcount(fptype.compute_fingerprint(mol2))
1
```

The non-thread-safe version uses `make_fingerprinter` to create a new fingerprinter function, which I've assigned to *calc_fingerprint*, and then call directly:

```
>>> calc_fingerprint = fptype.make_fingerprinter()
>>> bitops.byte_popcount(calc_fingerprint(mol1))
12
>>> bitops.byte_popcount(calc_fingerprint(mol2))
1
```

The keen-eyed will note that I could have written the first code as:

```
>>> compute_fingerprint = fptype.compute_fingerprint
>>> bitops.byte_popcount(compute_fingerprint(mol1))
12
>>> bitops.byte_popcount(compute_fingerprint(mol2))
1
```

and gotten the same answer, which means there is little API need for a special "make_fingerprinter()" function, except for performance.

I timed the performance. Even in the worst case that I could find (Open Babel's FP2 fingerprint), the performance boost was a paltry 2.5%. Otherwise it was about 1%. This is not enough to warrant using this function.

(Why do I leave it in? Probably because of the hard work I put into writing it, and because I like the principle behind it. Perhaps I have hopes that the performance difference will be more apparent on multi-threaded benchmarks, which I haven't evaluated.)

Toolkit API examples

This chapter gives many examples of how to use the toolkit API. For an overview of the toolkit API functions, see *chemfp.toolkit*. For details about actual toolkit implementations, see *chemfp.openeye_toolkit*, *chemfp.openbabel_toolkit*, *chemfp.rdkit_toolkit*, and *chemfp.text_toolkit*.

Fingerprint search usually starts with a structure record, and not a fingerprint. The functions *chemfp.read_molecule_fingerprints()* and *chemfp.read_molecule_fingerprints_from_string()* give a quick way to read a file or string containing structure records as the corresponding fingerprints.

Sometimes you want more control over the process. You might want to generate multiple fingerprints for the same structure and not want to reparse the structure record multiple times. Or you might want to return the search results as extra fields to the query SDF record instead of a simple list of values.

Chemfp uses a third-party chemistry toolkit to parse the records into a molecule, or compute the fingerprint for a given molecule. It's not hard to write your own Open Babel, OEChem/OEGraphSim, or RDKit code to handle any of these or similar tasks. The problem comes in when you want to mix multiple fingerprint types, like to compare the default RDKit fingerprint to Open Babel's FP2 fingerprint. You end up writing very different code for essentially the same fingerprint task.

There's an old saying in computer science; all problems can be solved with another layer of indirection. The chemfp toolkit API follows that tradition. It's a common API for molecule I/O which works across the three supported toolkits. It's also my best effort at implementing a next generation API.

Bear in mind that it is only an *I/O* API. Chemfp is a fingerprint toolkit and it will not gain a common molecule API. For that, look toward Cinfony.

Get a chemfp toolkit

In this section you'll learn how to load a "toolkit" – a portable API layer above the actual chemistry toolkit – and how to check if a toolkit is available and has a valid license.

Each toolkit I/O adapter is implemented as a chemfp submodule. If you know the underlying chemistry toolkit is installed you can import the adapter directly:

```
>>> from chemfp import openbabel_toolkit
>>> from chemfp import openeye_toolkit
>>> from chemfp import rdkit_toolkit
```

Use *chemfp.get_toolkit_names()* to get the available toolkit names:

```
>>> chemfp.get_toolkit_names()
set(['openeye', 'rdkit', 'openbabel'])
```

This will try to import each module, which means it may take a second or more depending on the shared library load time for your system. (This overhead only occurs once.) The function returns a list of the modules that could be loaded and have a valid license.

You can use *chemfp.get_toolkit()* to get the correct toolkit module given a name; it raises a ValueError if the underlying toolkit isn't installed or the toolkit name is unknown:

```
>>> chemfp.get_toolkit("rdkit")
<module 'chemfp.rdkit_toolkit' from 'chemfp/rdkit_toolkit.pyc'>
>>> chemfp.get_toolkit("openeye")
<module 'chemfp.openeye_toolkit' from 'chemfp/openeye_toolkit.pyc'>
>>> chemfp.get_toolkit("openbabel")
<module 'chemfp.openbabel_toolkit' from 'chemfp/openbabel_toolkit.pyc'>
```

Existence isn't enough to know if you can use a toolkit. OEChem requires a license. Each I/O adapter implements *chemfp.toolkit.is_licensed()*. It returns True for Open Babel and RDKit and the value of OEChemIsLicensed() for OEChem:

```
>>> from __future__ import print_function
>>> for name in chemfp.get_toolkit_names():
...   T = chemfp.get_toolkit(name)
...   print("Toolkit %r (%s) is licensed? %s" % (T.name, T.software, T.is_licensed()))
...
Toolkit 'openeye' (OEChem/20170208) is licensed? True
Toolkit 'rdkit' (RDKit/2016.09.3) is licensed? True
Toolkit 'openbabel' (OpenBabel/2.4.1) is licensed? True
```

(Thanks OpenEye for an no-cost developer license to their toolkit!) There is currently no way to check if OEGraphSim is licensed; you'll need to use native OpenEye code instead.

For fun I also showed the `software` attribute, which gives more detailed information about the toolkit version in a standardized format.

Finally, use *chemfp.has_toolkit()* to check if a toolkit is available. In the following, I used one of my local testing environments which has OEChem installed but not the other toolkits. (I use virtualenv to create and manage these environments; it's a very useful tool.):

```
>>> chemfp.has_toolkit("openeye")
True
>>> chemfp.has_toolkit("openbabel")
False
>>> chemfp.has_toolkit("rdkit")
False
```

The other option is to catch the ValueError raised when trying to get an unavailable toolkit:

```
>>> chemfp.get_toolkit("openeye")
<module 'chemfp.openeye_toolkit' from 'chemfp/openeye_toolkit.py'>
>>> chemfp.get_toolkit("rdkit")
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "chemfp/__init__.py", line 1823, in get_toolkit
    raise ValueError("Unable to get toolkit %r: %s" % (toolkit_name, err))
ValueError: Unable to get toolkit 'rdkit': No module named rdkit
```

```
>>> chemfp.get_toolkit("cdk")
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "chemfp/__init__.py", line 1845, in get_toolkit
    raise ValueError("Toolkit %r is not supported" % (toolkit_name,))
ValueError: Toolkit 'cdk' is not supported
```

This is a bit more complicated to do, but does have the advantage of giving access to an error message.

Parse and create SMILES

In this section you'll learn how to parse a SMILES into a molecule, convert a molecule into a SMILES, and the difference between a SMILES record and a SMILES string. You will need a chemistry toolkit for this and most of the examples in this chapter.

The chemfp toolkit I/O API is the same across the different toolkits, and examples with one will generally work with the other, except for essential differences like the specific formats available, the chemistry differences in how to interpret a record, the error messages, and reader and writer arguments.

For most examples I'll use `T` as the toolkit module name, rather than specify a specific toolkit. My examples will be based on RDKit, but you can use any of the following, if available on your system:

```
# Choose one of these
from chemfp import openeye_toolkit as T
from chemfp import openbabel_toolkit as T
from chemfp import rdkit_toolkit as T
```

I'll parse the SMILES string for phenol as a toolkit molecule, then convert the toolkit molecule into its canonical isomeric SMILES string using *chemfp.toolkit.create_string()*:

```
>>> mol = T.parse_molecule("c1ccccc1O", "smistring")
>>> mol
<rdkit.Chem.rdchem.Mol object at 0x103559980>
>>> T.create_string(mol, "smistring")
'Oc1ccccc1'
```

The "smistring" format name means that the input is a SMILES string. Chemfp follows the rule from the original SMILES paper that the SMILES string ends at the first whitespace. The following is valid across the chemfp toolkits API even if the underlying toolkit doesn't accept the "junk" as part of a SMILES:

```
>>> mol = T.parse_molecule("c1ccccc1O junk", "smistring")
```

On the other hand, if you have a SMILES record, which is a SMILES string followed by an id and possibly other fields, then use the "smi" format name. That will parse the first characters as a SMILES string and parse the rest of the input, up to the end of the line, as the record id:

```
>>> mol = T.parse_molecule("c1ccccc1O junk", "smistring")
>>> T.get_id(mol)
>>> mol = T.parse_molecule("c1ccccc1O junk", "smi")
>>> T.get_id(mol)
'junk'
>>> mol = T.parse_molecule("c1ccccc1O flotsam and jetsam\nand more\n", "smi")
>>> T.get_id(mol)
'flotsam and jetsam'
```

I used the *chemfp.toolkit.get_id()* helper function. While chemfp doesn't have a common molecule object, I found I do need a few standard functions to manipulate toolkit molecules. Also, *toolkit.parse_molecule()* will only read the first record and ignore trailing data, which is why the "and more" didn't affect anything.

Now that the molecule has an id, it's easy to see the difference between the "smistring" and "smi" in the output string:

```
>>> T.create_string(mol, "smistring")
'Oc1ccccc1'
>>> T.create_string(mol, "smi")
'Oc1ccccc1 flotsam and jetsam\n'
```

Finally, you can pass an alternate *id* to the *toolkit.create_string()* function. One example of when this is useful is when your identifier comes from one field of a database and the SMILES string from another, and you want to combine the results to get an SDF record:

```
>>> T.create_string(mol, "smi", id="nothing to see here")
'Oc1ccccc1 nothing to see here\n'
```

WARNING: The toolkit implementation may temporarily change then restore the toolkit molecule's own identifier in order to get the correct output. This is not thread-safe.

Canonical, non-isomeric, and arbitrary SMILES

In this section you'll learn the difference between the "smistring", "canstring", and "usmstring" SMILES string formats and the "smi", "can", and "usm" SMILES record formats. As with all examples which use the generic `T` toolkit name, you'll need one of the supported chemistry toolkits, and I'll use RDKit as my underlying toolkit.

The SMILES format supports many different ways to represent the same molecule. "CO", "OC", "[OH][CH3]", and "C3.O3" are four different SMILES strings for methanol. A canonicalization algorithm uses additional rules to create a unique SMILES representation for a given molecular graph. The different chemistry toolkit have different canonicalization algorithms, so each toolkit will likely generate a different canonical SMILES string for the same molecular graph.

There are multiple classes of canonical SMILES strings even in the same toolkit. The original SMILES format did not handle isotopes, chirality, or stereochemistry. The later extension to support these was called "isomeric SMILES", to distinguish it from the original SMILES.

Because of the history, when people asked a toolkit for "SMILES" output they got non-isomeric non-canonical SMILES, while "canonical SMILES" gave them "non-isomeric canonical". This caused subtle usability errors. Many people, including people like me who should have the experience to know better, expect canonical isomeric SMILES by default. But for over 20 years all of the toolkits followed Daylight's lead in how they did things.

OEChem 2.0 broke with tradition and fixed the mistake. It defined the default SMILES as canonical isomeric SMILES. If you make the effort to ask for a canonical SMILES you get canonical non-isomeric SMILES, and if you really want non-canonical, non-isomeric SMILES you can ask for the "usm" format.

Chemfp follows OpenEye's lead. The "smistring" format generates a canonical isomeric SMILES string, the "canstring" format generates a canonical non-isomeric SMILES string, and the "usmstring" format generates a non-canonical non-isomeric SMILES string:

```
>>> from chemfp import rdkit_toolkit as T  # use your toolkit of choice
>>>
>>> mol = T.parse_molecule("[235P].[238U]", "smistring")
>>> T.create_string(mol, "smistring")
u'[235P].[238U]'
>>> T.create_string(mol, "canstring")
u'[P].[U]'
```

```
>>> T.create_string(mol, "usmstring")
u'[P].[U]'
```

Here's evidence that the "usmstring" format is non-canonical:

```
>>> mol = T.parse_molecule("[238U].[235P]", "smistring")
>>> T.create_string(mol, "usmstring")
u'[U].[P]'
>>> T.create_string(mol, "smistring")
u'[235P].[238U]'
```

These conventions also apply when creating "smi", "can", and "usm" strings:

```
>>> T.set_id(mol, "radioactive")
>>> T.create_string(mol, "smi")
u'[235P].[238U] radioactive\n'
>>> T.create_string(mol, "can")
u'[P].[U] radioactive\n'
>>> T.create_string(mol, "usm")
u'[U].[P] radioactive\n'
```

By the way, *chemfp.toolkit.parse_molecule()* doesn't distinguish between "smi", "can" and "usm" as input SMILES records, nor between "smistring", "canstring" and "usmstring". The format only makes a difference for output. Later on you'll see how to specify *writer_args* to have more fine-grained control over the output SMILES format. (See *RDKit-specific SMILES reader_args and writer_args*, *OpenEye-specific SMILES reader_args and writer_args*, and *Open Babel-specific SMILES reader_args and writer_args* for toolkit-specific examples.)

Use *format* to create a record in SDF format

In this section you'll learn how to convert a toolkit molecule into an SDF record. This example will use the RDKit toolkit but the results will be substantially the same for any of the three supported chemistry toolkits.

To create an SDF record as a Unicode string, pass "sdf" as the *format* to *chemfp.toolkit.create_string()*:

```
>>> from __future__ import print_function
>>> from chemfp import rdkit_toolkit as T # use your toolkit of choice
>>> mol = T.parse_molecule("CO", "smistring")
>>> print(T.create_string(mol, "sdf"))

     RDKit

  2  1  0  0  0  0  0  0  0  0999 V2000
    0.0000    0.0000    0.0000 C   0  0  0  0  0  0  0  0  0  0  0  0
    0.0000    0.0000    0.0000 O   0  0  0  0  0  0  0  0  0  0  0  0
  1  2  1  0
M  END
$$$$
```

Starting with chemfp 3.0, the `create_string()` function returns a Unicode string, under both Python 2.7 and Python 3.5+:

```
>>> T.create_string(mol, "sdf")[:13]
u'\n     RDKit  '
```

In earlier versions of chemfp, `create_string()` returned a byte string. This was the usual practice under Python 2.5 to 2.7. It was fine for ASCII data, but caused problems with other characters, like Greek letters in a compound

name or a data item listing prices in with the GBP or EUR symbol.

Python 3 makes a strong distinction between a byte string and a Unicode string. Chemfp 3.x follows that lead by having `create_string()` return a Unicode string, and added the new function *chemfp.toolkit.create_bytes()* to return a byte string:

```
>>> T.create_bytes(mol, "sdf")[:13]
'\n     RDKit  '
```

Here I'll set the molecule's name to the lower-case Greek letter 'alpha', and show you the interactive output from Python 2.7:

```
>>> T.set_id(mol, u"\N{GREEK SMALL LETTER ALPHA}")
>>> T.create_string(mol, "sdf")[:13]
u'\u03b1\n     RDKit '
>>> T.create_bytes(mol, "sdf")[:13]
'\xce\xb1\n     RDKit'
>>> print(T.create_string(mol, "sdf")[:13])
α
     RDKit
```

Here's the same output under Python 3.5:

```
>>> T.set_id(mol, u"\N{GREEK SMALL LETTER ALPHA}")
>>> T.create_string(mol, "sdf")[:13]
'α\n     RDKit '
>>> T.create_bytes(mol, "sdf")[:13]
b'\xce\xb1\n     RDKit'
>>> print(T.create_string(mol, "sdf")[:13])
α
     RDKit
```

Use zlib record compression

In this section you'll learn about the "zlib" compression option for single record parsers and byte string creation.

A record in SDF format can be large, but most of the content is repetetive. Often it's better to store a zlib compressed record in a database instead of the full record. When I use zlib to compress each record of Compound_027575001_027600000.sdf.gz I get a 4.5-fold compression. That is, the uncompressed records take 2704906 bytes, the individually compressed records take 594682 bytes, and the gzip compressed file takes 270419. (Gzip is nearly twice as good as individually compressed records because it can collect compression statistics across multiple records and build a better prediction model.)

Chemfp supports a zlib compression option for the record-oriented functions, though not the file-oriented functions. To enable it, add ".zlib" to the format string for *chemfp.toolkit.create_bytes()*. Here you can see how adding that suffix reduces the record size:

```
>>> from __future__ import print_function
>>> from chemfp import rdkit_toolkit as T # use your toolkit of choice
>>> mol = T.parse_molecule("CO", "smistring")
>>> print("uncompressed:", len(T.create_bytes(mol, "sdf")))
uncompressed: 228
>>> print("compressed:", len(T.create_bytes(mol, "sdf.zlib")))
compressed: 77
```

I'll complete a round-trip conversion by parsing the compressed SD record to a molecule then converting it to a SMILES string:

```
>>> compressed = T.create_bytes(mol, "sdf.zlib")
>>> new_mol = T.parse_molecule(compressed, "sdf.zlib")
>>> T.create_string(new_mol, "smistring")
'CO'
```

The zlib option only works with `create_bytes`; it does not work with `create_string` because the latter only returns Unicode strings, and it's possible for zlib to return something which isn't valid Unicode. Here's what happens if you try to use it anyway:

```
>>> T.create_string(mol, "sdf.zlib")
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "chemfp/rdkit_toolkit.py", line 428, in create_string
    return _toolkit.create_string(mol, format, id, writer_args, errors)
  File "chemfp/base_toolkit.py", line 1333, in create_string
    raise ValueError("create_string() does not support compression. Use create_bytes()
↪")
ValueError: create_string() does not support compression. Use create_bytes()
```

On the other hand, *chemfp.toolkit.parse_molecule()* takes both Unicode strings and byte strings as input. It treats the byte strings as being UTF-8 encoded.

Get a list of available formats and distinguish between input and output formats

In this section you'll learn how to get the list of available formats for each object, and determine if a format can be used to get a toolkit molecule from a string record, or convert a toolkit molecule into a string record.

The toolkit's *chemfp.toolkit.get_formats()* function returns a list of the available formats. On my computer RDKit supports 13 formats, OEChem 24, and Open Babel (showing off its heritage) supports a whopping 189:

```
>>> from chemfp import rdkit_toolkit
>>> len(rdkit_toolkit.get_formats())
13
>>> rdkit_toolkit.get_formats()
[Format('rdkit/smi'), Format('rdkit/can'), Format('rdkit/usm'),
Format('rdkit/sdf'), Format('rdkit/smistring'),
Format('rdkit/canstring'), Format('rdkit/usmstring'),
Format('rdkit/molfile'), Format('rdkit/rdbinmol'),
Format('rdkit/inchi'), Format('rdkit/inchikey'),
Format('rdkit/inchistring'), Format('rdkit/inchikeystring')]
>>>
>>> from chemfp import openeye_toolkit
>>> len(openeye_toolkit.get_formats())
24
>>> openeye_toolkit.get_formats()
[Format('openeye/smi'), Format('openeye/usm'),
Format('openeye/can'), Format('openeye/sdf'),
Format('openeye/molfile'), Format('openeye/skc'),
Format('openeye/mol2'), Format('openeye/mol2h'),
Format('openeye/sln'), Format('openeye/mmod'),
Format('openeye/pdb'), Format('openeye/xyz'), Format('openeye/cdx'),
Format('openeye/mopac'), Format('openeye/mf'),
Format('openeye/oeb'), Format('openeye/inchi'),
Format('openeye/inchikey'), Format('openeye/smistring'),
Format('openeye/canstring'), Format('openeye/usmstring'),
Format('openeye/slnstring'), Format('openeye/inchistring'),
Format('openeye/inchikeystring')]
```

```
>>>
>>> from chemfp import openbabel_toolkit
>>> len(openbabel_toolkit.get_formats())
189
>>> openbabel_toolkit.get_formats()
[Format('openbabel/smi'), Format('openbabel/can'),
Format('openbabel/usm'), Format('openbabel/smistring'),
Format('openbabel/canstring'), Format('openbabel/usmstring'),
Format('openbabel/sdf'), Format('openbabel/inchi'),
Format('openbabel/inchikey'), Format('openbabel/inchistring'),
Format('openbabel/inchikeystring'), Format('openbabel/mp'),
Format('openbabel/gzmat'), Format('openbabel/txt'),
   ... many formats omitted ...
Format('openbabel/pdb')]
>>>
```

I'll use *chemfp.toolkit.get_format()*, which returns a *chemfp.base_toolkit.Format*, to get the "sdf" format for OpenEye:

```
>>> sdf_format = openeye_toolkit.get_format("sdf")
>>> sdf_format.name
'sdf'
>>> sdf_format.toolkit_name
'openeye'
```

The "sdf" format can be used for both input and output in all toolkits:

```
>>> sdf_format.is_input_format, sdf_format.is_output_format
(True, True)
```

However, some formats are output only, like the InChIKey format (assuming it's available for your toolkit):

```
>>> inchi_fmt = openeye_toolkit.get_format("inchikey")
>>> inchi_fmt.is_input_format, inchi_fmt.is_output_format
(False, True)
```

On the other hand, some formats are input only, like Open Babel's support for MOPAC's output format:

```
>>> mopout_fmt = openbabel_toolkit.get_format("mopout")
>>> mopout_fmt.is_input_format, mopout_fmt.is_output_format
(True, False)
```

Instead of asking for all available formats, you can ask for only the input formats, or only the output formats, using *chemfp.toolkit.get_input_formats* or *chemfp.toolkit.get_output_formats*:

```
>>> from __future__ import print_function
>>> import chemfp
>>> for toolkit_name in ("openbabel", "openeye", "rdkit"):
...   T = chemfp.get_toolkit(toolkit_name)
...   print(toolkit_name, "has", len(T.get_input_formats()), "input formats")
...   print(toolkit_name, "has", len(T.get_output_formats()), "output formats")
...
openbabel has 147 input formats
openbabel has 138 output formats
openeye has 16 input formats
openeye has 23 output formats
rdkit has 11 input formats
rdkit has 13 output formats
```

Determine the format for a given filename

It's sometimes useful to know what format will be used for a given filename. A filename can be used as a source for a reader or destination for a writer, and a toolkit might understand a given format when used as input but not as ouput, or vice-versa.

The function *chemfp.toolkit.get_input_format_from_source()* returns a *chemfp.base_toolkit.Format* for the given filename:

```
>>> from chemfp import rdkit_toolkit as T # use your toolkit of choice
>>> T.get_input_format_from_source("abc.smi.gz")
Format('rdkit/smi.gz')
```

This is the same Format object you saw in the previous section. I didn't mention the `compression` attribute in that discussion. It's "gz" for gzip-ed files, and the empty string "" for uncompressed files.

```
>>> fmt = T.get_input_format_from_source("abc.smi.gz")
>>> fmt.name
'smi'
>>> fmt.compression
'gz'
>>>
>>> fmt = T.get_input_format_from_source("abc.smi")
>>> fmt.name
'smi'
>>> fmt.compression
''
```

Asking for a supported format which isn't an input format raises a ValueError exception:

```
>>> from chemfp import openbabel_toolkit
>>> openbabel_toolkit.get_input_format_from_source("example.inchikey")
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "chemfp/openbabel_toolkit.py", line 143, in get_input_format_from_source
    return _format_registry.get_input_format_from_source(source, format)
  File "chemfp/base_toolkit.py", line 841, in get_input_format_from_source
    format_config = self.get_input_format_config(register_name)
  File "chemfp/base_toolkit.py", line 765, in get_input_format_config
    % (self.external_name, register_name))
ValueError: Open Babel does not support 'inchikey' as an input format
```

even though "inchikey" is supported as an output format:

```
>>> openbabel_toolkit.get_output_format_from_destination("example.inchikey")
Format('openbabel/inchikey')
```

Yes, there's a different function to get the format name for a source filename than for a destination filename. Maybe in the future I'll support a generic `get_format_from_filename()`; let me know if that would be useful.

If you ask for a format which doesn't exist then the functions raises a different ValueError exception:

```
>>> openbabel_toolkit.get_input_format_from_source("example.does-not-exist")
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
```

```
    .....
  File "/Library/Python/2.7/site-packages/chemfp/base_toolkit.py", line 532, in get_
↪format_config
    % (self.external_name, register_name))
ValueError: Open Babel does not support the 'does-not-exist' format
```

I've found it useful to have a way to override the default guess. It's amazing how many people use ".dat" for SMILES or SDF files, and ".txt" files for SMILES. The format lookup functions support a second, optional parameter, which is the format name to use.

```
>>> openbabel_toolkit.get_input_format_from_source("example.does-not-exist", "smi.gz")
Format('openbabel/smi.gz')
```

This exists so that code like:

```
if format is not None:
    fmt = T.get_format(format)
else:
    fmt = T.get_format_from_source(filename)
```

can be replaced with:

```
fmt = T.get_format_from_source(filename, format)
```

Working with a format object is useful when combined with format's *reader_args* and *writer_arg* functions discussed in *Specify a SMILES delimiter through reader_args*

```
>>> fmt = openbabel_toolkit.get_input_format_from_source("input.smi")
>>> fmt.get_default_writer_args()
{'explicit_hydrogens': False, 'isomeric': True, 'delimiter': None, 'options': None,
↪'canonicalization': 'default'}
>>> fmt.get_writer_args_from_text_settings({
...     "explicit_hydrogens": "true",
...     "isomeric": "false",
...     "delimiter": "tab"})
{'explicit_hydrogens': True, 'isomeric': False, 'delimiter': 'tab'}
```

Parse the id and the molecule at the same time

In this section you'll learn how to parse a structure record, as a string, to extract both the identifier and the native molecule object.

Usually you want both the molecule and its id. You could parse the molecule then use *T.get_id(mol)* to get the id, but that's extra work, it leads to awkward looking code, and is slower than having chemfp do the work for you when it parses the molecule.

Instead, use *chemfp.toolkit.parse_id_and_molecule()*:

```
>>> from chemfp import rdkit_toolkit as T  # use your toolkit of choice
>>>
>>> T.parse_id_and_molecule("C1C(C)=C(C=CC(C)=CC=CC(C)=CCO)C(C)(C)C1 vitamin a", "smi
↪")
(u'vitamin a', <rdkit.Chem.rdchem.Mol object at 0x1035f14b0>)
```

Note that the identifier is a Unicode string. This is new with chemfp 3.0. Earlier versions returned byte string instead.

If there is no id/title field then the id will either be None or the empty string, depending on the toolkit and format:

```
>>> T.parse_id_and_molecule("C", "smi")
(None, <rdkit.Chem.rdchem.Mol object at 0x1035f14b0>)
```

Instead of testing for the empty string or None, your code you should use "if not id:" to test for a missing id:

```
>>> id, mol = T.parse_id_and_molecule("C", "smi")
>>> if not id:
...   print("Missing id!")
...
Missing id!
```

Specify alternate error behavior

In this section you'll learn how to use the *errors* parameter to have *chemfp.toolkit.parse_molecule()* return None rather than raise an exception, and to have it print a report about the failing molecule.

The string "Q" is not a valid SMILES string. All of the toolkits will fail to parse it, and the chemfp toolkit I/O adapter by default raises an exception when that happens:

```
>>> from chemfp import openbabel_toolkit
>>> openbabel_toolkit.parse_molecule("Q", "smistring")
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
      ...
  File "chemfp/io.py", line 87, in error
    _compat.raise_tb(ParseError(msg, location), None)
  File "<string>", line 1, in raise_tb
chemfp.ParseError: Open Babel cannot parse the SMILES 'Q'
>>>
>>> rdkit_toolkit.parse_molecule("Q", "smistring")
[01:14:30] SMILES Parse Error: syntax error for input: 'Q'
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
    ...
  File "chemfp/io.py", line 87, in error
    _compat.raise_tb(ParseError(msg, location), None)
  File "<string>", line 1, in raise_tb
chemfp.ParseError: RDKit cannot parse the SMILES string 'Q'
>>>
>>> from chemfp import openeye_toolkit
>>> openeye_toolkit.parse_molecule("Q", "smistring")
Warning: Problem parsing SMILES:
Warning: Q
Warning: ^

Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
   ...
  File "chemfp/io.py", line 87, in error
    _compat.raise_tb(ParseError(msg, location), None)
  File "<string>", line 1, in raise_tb
chemfp.ParseError: OEChem cannot parse the smistring record: 'Q'
```

On the other hand, "[NH8]" is a valid SMILES, but RDKit by default will reject it as chemically unreasonable, while OEChem and Open Babel are less strict and treat it as a molecular graph rather than a chemical molecule.

I'll write a program which checks which toolkits will parse "[NH8]"

```
# I call this "check_NH8.py"
from __future__ import print_function
import chemfp
allowed = []; rejected = []
for name in chemfp.get_toolkit_names():
  T = chemfp.get_toolkit(name)
  try:
    T.parse_molecule("[NH8]", "smistring")
  except ValueError:
    rejected.append(name)
  else:
    allowed.append(name)
print("Allowed:", allowed, "Rejected:", rejected)
```

```
% python check_NH8.py
[01:25:49] Explicit valence for atom # 0 N, 8, is greater than permitted
Allowed: ['openeye', 'openbabel'] Rejected: ['rdkit']
```

I think the try/except/else is sometimes harder to understand than returning an error value, because it's harder to see the control flow. I can ask *chemfp.toolkit.parse_molecule()* to `ignore` errors, which causes it to return a None object rather than raise an exception. turns the above loop into the following:

```
for name in chemfp.get_toolkit_names():
  T = chemfp.get_toolkit(name)
  mol = T.parse_molecule("[NH8]", "smistring", errors="ignore")
  if mol is None:
    rejected.append(name)
  else:
    allowed.append(name)
```

The *errors* option is more useful in later sections, when parsing multiple records.

The `errors` parameter can also take the value `report`. Like `ignore`, this will return a None when there is an error rather than raise an exception. It will also write a consistent, cross-toolkit error message to stderr, including the SMILES string that failed if the input is a SMILES:

```
>>> for name in chemfp.get_toolkit_names():
...   T = chemfp.get_toolkit(name)
...   T.parse_molecule("Q", "smistring", errors="report")
...   T.parse_molecule("[NH8]", "smistring", errors="report")
...
Warning: Problem parsing SMILES:
Warning: Q
Warning: ^

ERROR: OEChem cannot parse the smistring record: 'Q'. Skipping.
<oechem.OEGraphMol; proxy of <Swig Object of type 'OEGraphMolWrapper *' at␣
↪0x104a9ffc0> >
[01:31:22] SMILES Parse Error: syntax error for input: 'Q'
ERROR: RDKit cannot parse the SMILES string 'Q'. Skipping.
[01:31:22] Explicit valence for atom # 0 N, 8, is greater than permitted
ERROR: RDKit cannot parse the SMILES string '[NH8]'. Skipping.
ERROR: Open Babel cannot parse the SMILES 'Q'. Skipping.
<openbabel.OBMol; proxy of <Swig Object of type 'OpenBabel::OBMol *' at 0x10756bab0> >
```

The *chemfp.toolkit.parse_id_and_molecule()* function also takes the `errors` parameter. If the structure could not be parsed then the second component of the tuple (the molecule) will be None. The first component (the id) may or or may not be None, depending on the underlying implementation:

```
>>> from chemfp import rdkit_toolkit
>>> rdkit_toolkit.parse_id_and_molecule("Q q-ane", "smi", errors="ignore")
[01:33:48] SMILES Parse Error: syntax error for input: 'Q'
(None, None)
>>>
>>> from chemfp import openeye_toolkit
>>> openeye_toolkit.parse_id_and_molecule("Q q-ane", "smi", errors="ignore")
Warning: Problem parsing SMILES:
Warning: Q q-ane
Warning: ^

(None, None)
>>>
>>> from chemfp import openbabel_toolkit
>>> openbabel_toolkit.parse_id_and_molecule("Q q-ane", "smi", errors="ignore")
('q-ane', None)
```

Future versions of chemfp may work to normalize this behavior, or let the caller choose a specific behavior.

Specify a SMILES delimiter through reader_args

In this section you'll learn how to parse a SMILES record as a set of delimited fields instead of the default of a SMILES string followed by a title, and some of the limitations of chemfp's attempt at a consistent cross-toolkit SMILES record parser.

You might think that the SMILES file format is well defined, but it sadly isn't. Different toolkits have slightly different interpretations for a SMILES record format. Consider the SMILES record:

```
C1C(C)=C(C=CC(C)=CC=CC(C)=CCO)C(C)(C)C1 vitamin a
```

The original Daylight definition is that a SMILES record is single line, which starts with the SMILES string. The SMILES string ends with the first whitespace character or the end of the line, and if there was a whitespace character than the rest of the line is the title. OpenEye follows this definition, as does chemfp. That's why the previous example extracted "vitamin A" as the record id.

However, RDKit treats a SMILES file record as a space or tab separated set of fields, where the first field is the SMILES, the second field is the id/title and additional columns may store other properties. RDKit would use "vitamin" as the record id for this record. (RDKit can also be configured to interpret the first line as column names. Chemfp does not currently support this option, though I plan to have a cross-platform implementation in a future release.)

Chemfp normalizes the SMILES record parser API so that all toolkits by default expect the Daylight format. Use the optional *reader_args* dictionary to specify an alternate interpretation:

```
>>> from chemfp import rdkit_toolkit as T  # use your toolkit of choice
>>>
>>> smiles = "C1C(C)=C(C=CC(C)=CC=CC(C)=CCO)C(C)(C)C1 vitamin a"
>>> T.parse_id_and_molecule(smiles, "smi", reader_args={"delimiter": "whitespace"})
(u'vitamin', <rdkit.Chem.rdchem.Mol object at 0x10f5ccfa0>)
```

In this case I asked it to parse the record as a set of whitespace delimited fields. If you have tab-separated fields, where a space inside of a field is not part of the delimiter, then use the "tab" delimiter:

```
>>> T.parse_id_and_molecule("O=O\tmolecular oxygen\t31.9988\n", "smi",
...                         reader_args={"delimiter": "tab"})
(u'molecular oxygen', <rdkit.Chem.rdchem.Mol object at 0x10fbe9590>)
```

The supported delimiters are:

- to-eol - (default) everything past the first whitespace is interpreted as the id/title;
- tab or "\t" - the fields are tab-separated; the first field is the SMILES and the second the id;
- space or " " - the fields are space-separated;
- whitespace - the fields are whitespace-separated;
- native - use the native interpretation for the given toolkit;

While chemfp strives for cross-toolkit portability, it is not perfect. Leading and trailing whitespace might not be supported, so the first character of the SMILES record must also be the first character of the SMILES string. Also, the toolkit is free to interpret the first whitespace as the delimiter despite the *reader_args* setting. You can see the difference between RDKit and OEChem toolkits in the following, where the record is tab-separated but OEChem always stops parsing a SMILES at the first whitespace:

```
>>> smiles = "O=O\tmolecular oxygen\n"
>>>
>>> from chemfp import openeye_toolkit
>>> openeye_toolkit.parse_id_and_molecule(smiles, "smi", reader_args={"delimiter":
↪"space"})
(u'molecular', <oechem.OEGraphMol; proxy of <Swig Object of type 'OEGraphMolWrapper *
↪' at 0x10fbe0090> >)
>>>
>>> from chemfp import rdkit_toolkit
>>> rdkit_toolkit.parse_id_and_molecule(smiles, "smi", reader_args={"delimiter":
↪"space"})
[01:38:59] SMILES Parse Error: syntax error for input: 'O=O   molecular'
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
    ...
  File "chemfp/io.py", line 87, in error
    _compat.raise_tb(ParseError(msg, location), None)
  File "<string>", line 1, in raise_tb
chemfp.ParseError: RDKit cannot parse the SMILES 'O=O\tmolecular'
```

Neither the SMILES parser nor the other parsers validate the full contents of the *reader_args* dictionary. Extra items are ignored. This is deliberate because it lets you combine, say, SMILES and SDF parameters in the same dictionary without needing to check the specific format first.

To a lesser extent, it also makes it easier to specify parameters which work across multiple toolkit versions. For example, the most recent version of OEChem's SMILES parsers added a *quiet* option, which chemfp will support in the future. Your code can have a {"quiet": True} without first checking to see if this version of chemfp is new enough to support the parameter.

WARNING: As a result, it's very easy to specify a key with a typo, which is ignored, and not notice that it nothing happens.

WARNING #2: Really, I've been bitten by this a few times. Be extra cautious to check that you are using the right keys.

Specify an output SMILES delimiter through writer_args

In this section you'll learn how to create a SMILES record with a tab character separating the SMILES from the title using the *writer_args* parameter of *chemfp.toolkit.create_string()*.

By default `create_string` uses a space character to separate the SMILES from the rest of the id:

```
>>> from chemfp import rdkit_toolkit as T  # use your toolkit of choice
>>>
>>> mol = T.parse_molecule("O=O molecular oxygen\n", "smi")
>>> T.create_string(mol, "smi")
u'O=O molecular oxygen\n'
```

To use a tab character instead, pass in a *writer_args* dictionary with a "delimiter" of "tab":

```
>>> T.create_string(mol, "smi", writer_args={"delimiter": "tab"})
u'O=O\tmolecular oxygen\n'
```

The *writer_args* delimiter also accepts "whitespace", "space", "to-eol" and the other values from *reader_args*. Only "tab" and "\t" will use a tab character as the delimiter; all of the the others will use a space character.

Neither the SMILES writer nor the other writers validate the full contents of the *writer_args* dictionary. Extra items are ignored. This is deliberate because it lets you combine, say, SMILES and SDF parameters in the same dictionary without needing to check the specific format first. It also makes it easier to specify parameters which work across multiple toolkit versions.

WARNING: As a result, it's very easy to specify a key with a typo, which is ignored, and not notice that it nothing happens.

WARNING #2: Really, I've been bitten by this a few times. Be extra cautious to check that you are using the right keys.

RDKit-specific SMILES reader_args and writer_args

In this section you'll learn how to pass toolkit-specific parameters to the RDKit toolkit functions to parse and create a SMILES string. You will need the RDKit toolkit.

Earlier I showed that RDKit by default does a sanitization check to verify that the input is correct.

```
>>> from chemfp import rdkit_toolkit
>>> mol = rdkit_toolkit.parse_molecule("[NH8]", "smistring", errors="ignore")
[01:46:17] Explicit valence for atom # 0 N, 8, is greater than permitted
>>> mol is None
True
```

The underlying RDKit code to parse a SMILES string, MolFromSmiles, takes a *sanitize* parameter. The default, True, tells it to do the sanitization step, while False disables it.

Use the *reader_args* dictionary to pass the sanitize parameter to the underlying toolkit function:

```
>>> mol = rdkit_toolkit.parse_molecule("[NH8]", "smistring", reader_args={"sanitize":␣
↪False})
>>> mol
<rdkit.Chem.rdchem.Mol object at 0x107590a60>
>>> from rdkit import Chem
>>> Chem.MolToSmiles(mol)
'[NH8]'
```

Use the *writer_args* dictionary to pass toolkit-specific parameters to RDKit's MolToSmiles:

```
>>> mol = rdkit_toolkit.parse_molecule("c1ccccc1[16OH]", "smistring")
>>> rdkit_toolkit.create_string(mol, "smistring")
u'[16OH]c1ccccc1'
>>> rdkit_toolkit.create_string(mol, "smistring",
...                  writer_args={"isomericSmiles": False})
```

```
u'Oc1ccccc1'
>>> rdkit_toolkit.create_string(mol, "smistring",
...                   writer_args={"kekuleSmiles": True, "allBondsExplicit": True})
u'[16OH]-C1:C:C:C:C:C:1'
```

See *Get the default reader_args or writer_args for a format* for a description of how to get the default reader and writer arguments for a given format, and use help(*rdkit_toolkit.read_molecules*) and help(*rdkit_toolkit.open_molecule_writer*) to get a more human-readable description.

OpenEye-specific SMILES reader_args and writer_args

In this section you'll learn how to pass toolkit-specific parameters to the OEChem toolkit functions to parse and create a SMILES string. You will need the OEChem toolkit. See the next section for specific details about aromaticity.

By default the OEChem SMILES parser is tolerant of bad SMILES. I believe it's too tolerant, because will gladly parse what I think are invalid SMILES, like "C-=C":

```
>>> from chemfp import openeye_toolkit
>>> mol = openeye_toolkit.parse_molecule("C-=C", "smistring")
>>> openeye_toolkit.create_string(mol, "smistring")
u'C=C'
```

The developers at OpenEye recognize that pedantic folk like me exist. The OEChem SMILES parser has a "strict" mode, which I can enable in chemfp through the "flavor" parameter of the *reader_args* dictionary:

```
>>> mol = openeye_toolkit.parse_molecule("C-=C", "smistring",
...                     reader_args={"flavor": "Strict"})
Warning: Problem parsing SMILES:
Warning: Bond without end atom.
Warning: C-=C
Warning:   ^

Traceback (most recent call last):
  File "<stdin>", line 2, in <module>
      .... lines omitted ....
  File "chemfp/io.py", line 87, in error
    _compat.raise_tb(ParseError(msg, location), None)
  File "<string>", line 1, in raise_tb
chemfp.ParseError: OEChem cannot parse the smistring record: 'C-=C'
```

The underlying OEParseSmiles() function takes the optional *strict* and *canon* parameters. Why does chemfp use the term "flavor"? Why the capitalization for "Strict"?

Historically the low-level OEChem functions took individual parameters, like the positional arguments *canon* and *strict*:

```
>>> mol = OEGraphMol()
>>> OEParseSmiles(mol, "C-=C", False, True)
Warning: Problem parsing SMILES:
Warning: Bond without end atom.
Warning: C-=C
Warning:   ^

False
```

(I wrote "historically" because more recent versions have format-specific options classes, like OEParseSmilesOptions for SMILES. These collect all of the configuration options into a single parameter, which is easier to pass around.)

On the other hand, the high-level molecule parsers take a single "flavor" integer value to specify the options for a given format. This flavor is usually expressed as the union of a set of bitmasks. I'll show how OEChem's Python API uses the flavor parameter.

The following OEChem code reads a SMILES file in the default non-strict mode (with no specified flavor):

```
% cat example.smi
C=-C bad
CCC good
% python
   ...
>>> from __future__ import print_function
>>> from openeye.oechem import *
>>> ifs = oemolistream("example.smi")
>>> for mol in ifs.GetOEGraphMols():
...   print(mol.GetTitle(), mol.NumAtoms())
...
bad 2
good 3
```

while the following sets the SMILES flavor to use "strict" mode:

```
>>> ifs = oemolistream("example.smi")
>>> ifs.SetFlavor(OEFormat_SMI, OEIFlavor_SMI_Strict)
True
>>> for mol in ifs.GetOEGraphMols():
...   print(mol.GetTitle(), mol.NumAtoms())
...
Warning: Problem parsing SMILES:
Warning: Bond without end atom.
Warning: C=-C bad
Warning:   ^

Warning: Error reading molecule "" in Canonical stereo SMILES format.
good 3
```

(You can see some terminology differences between me and OpenEye in the warning message. The "Canonical" and "stereo" are only meaningful as a description of the output format, not the input format, and I use the traditional term "isomeric" while they highlight the more important stereochemistry aspect. I also got confused because I thought at first the "Canonical" had something to do with OEIFlavor_SMI_Canon.)

I decided to base *chemfp openeye_toolkit* API on the high-level "flavor" API of OEChem, which is better documented and requires less work on my part to implement than low-level functions. But I also decided to extend it to support a string value, and not just an integer.

To explain how that works, I'll switch from describing *reader_args* to *writer_args*, because raising an exception with the "Strict" option gets boring, fast.

The OEChem SMILES output flavors are: `OEOFlavor_SMI_AtomMaps`, `OEOFlavor_SMI_AtomStereo`, ... and you know what? The `OEOFlavor_SMI_` prefix is part of what makes the flavors hard to use in Python, so I'll omit the prefix in chemfp. The OEChem SMILES output flavors are: `AtomMaps`, `AtomStero`, `BondStereo`, `Canonical`, `ExtBonds`, `Hydrogens`, `ImpHCount`, `Isotopes`, `Kekule`, `RGroups`, `SmiMask`, and `SuperAtoms`. There are also `Default` and `DEFAULT` which are the bitwise union `RGroups|Isotopes|AtomStereo|BondStereo|AtomMaps|Canonical`.

In chemfp you can specify the fields as a "|" or "," separated list of flavor flags, without the prefix. Here are several different ways to specify the default settings for isomeric canonical SMILES string output:

```
>>> mol = openeye_toolkit.parse_molecule("[16O][*:1]", "smistring")
>>> openeye_toolkit.create_string(mol, "smistring")
u'[R1][16O]'
>>> openeye_toolkit.create_string(mol, "smistring",
...     writer_args={"flavor": ""})
u'[R1][16O]'
>>> openeye_toolkit.create_string(mol, "smistring",
...     writer_args={"flavor": "Default"})
u'[R1][16O]'
>>> openeye_toolkit.create_string(mol, "smistring",
...    writer_args={"flavor":
↪"RGroups|Isotopes|AtomStereo|BondStereo|AtomMaps|Canonical"})
u'[R1][16O]'
```

These settings override any options which might be implied by the format name. Thus, even though "smistring" is supposed to generate an isomeric canonical SMILES, I can use the *writer_args* to remove the isomeric component from the flavor:

```
>>> openeye_toolkit.create_string(mol, "smistring",
...    writer_args={"flavor": "RGroups|AtomStereo|BondStereo|AtomMaps|Canonical"})
u'[R1][O]'
```

While I used "|" as the separator, I can equally use ",", as in:

```
>>> openeye_toolkit.create_string(mol, "smistring",
...     writer_args={"flavor": "Isotopes,Canonical"})
'*[16O]'
```

OEChem uses the bar as a bitwise-or operator which merges the different flags. I added the comma as an alternative to the vertical bar because chemfp has additional syntax for removing options. The following removes the "RGroups" option from the isomeric and non-isomerical formats defaults, but otherwise leaves the defaults alone:

```
>>> openeye_toolkit.create_string(mol, "smistring",
...         writer_args={"flavor": "Default,-RGroups"})
u'[*:1][16O]'
>>>
>>> openeye_toolkit.create_string(mol, "canstring",
...         writer_args={"flavor": "Default,-RGroups"})
u'[*:1][O]'
```

(The terms are evaluated from left to right, so you can delete a term then add it back if you want.)

Writing this as `Default|-RGroups` caused the C programmer mind in me to gasp in bewilderment. ("The bitwise-or with the negative of the RGroups bitflags?!!")

You don't need to specify the OEChem flavor using a flavor string. You can also specify it as an integer:

```
>>> from openeye.oechem import *
>>> (OEOFlavor_SMI_Isotopes|OEOFlavor_SMI_AtomStereo|OEOFlavor_SMI_BondStereo|
...  OEOFlavor_SMI_AtomMaps|OEOFlavor_SMI_Canonical)
121
>>> openeye_toolkit.create_string(mol, "smistring",
...         writer_args={"flavor": 121})
u'[*:1][16O]'
>>> openeye_toolkit.create_string(mol, "smistring",
...         writer_args={"flavor": 0})
u'[O]*'
```

or (and this might be a bit excessive) as a string-encoded integer:

```
>>> openeye_toolkit.create_string(mol, "smistring",
...         writer_args={"flavor": "121"})
u'[*:1][16O]'
>>> openeye_toolkit.create_string(mol, "smistring",
...         writer_args={"flavor": "0"})
u'[O]*'
```

Chemfp tries to be helpful. It will include the list of available flavor names in the exception if it doesn't understand what you gave it:

```
>>> openeye_toolkit.create_string(mol, "smistring",
...         writer_args={"flavor": "chocolate"})
Traceback (most recent call last):
  File "<stdin>", line 2, in <module>
File "chemfp/openeye_toolkit.py", line 428, in create_string
  return _toolkit.create_string(mol, format, id, writer_args, errors)
         ...
  File "chemfp/_openeye_toolkit.py", line 895, in parse_flavor
    % (self.register_name, term, available_flavors))
ValueError: OEChem smi format does not support the 'chocolate'
flavor option. Available flavors are: AllBonds, AtomMaps,
AtomStereo, BondStereo, Canonical, ExtBonds, Hydrogens,
ImpHCount, Isotopes, Kekule, RGroups, SuperAtoms
```

See *Get the default reader_args or writer_args for a format* for a description of how to get the default reader and writer arguments for a given format, and use help(*openeye_toolkit.read_molecules*) and help(*openeye_toolkit.open_molecule_writer*) to get a more human-readable description.

OpenEye-specific aromaticity

In this section you'll learn how chemfp handles OpenEye's aromaticity parameter. You will need the OEChem toolkit, and you should read the previous section to understand some of the terminology.

Note: the OEGraphSim fingerprints are not affected by the *aromaticity* of the reader because they ensure that the molecules are always perceived using "openeye" aromaticity before generating the fingerprint.

The OpenEye toolkit supports the "openeye", "daylight", "tripos", "mdl", and "mmff" aromaticity models. In the high-level API, which is meant for reading and writing files or file-like objects, the aromaticity is an aspect of the flavor integer. If unspecified, OEChem uses the appropriate default aromaticity model for that format. As a result, aromaticity perception is required for both reading and writing files.

The low-level API handles file processing and aromaticity perception as distinct steps. This API can also process a single record directly, while the high-level API requires wrapping the record in a file-like object and then reading the first molecule from it.

The chemfp toolkit API is a high-level API for both files and records, which means I had to implement record conversion routines on top of OEChem's low-level API. Consequently, some of the details are different between the file I/O and record I/O APIs; the most significant being that the record I/O routines also support a "none" aromaticity.

The following shows the default aromaticity proceessing in action:

```
>>> from chemfp import openeye_toolkit
>>> mol = openeye_toolkit.parse_molecule("C1=CC=CC=C1", "smistring")
>>> [bond.IsAromatic() for bond in mol.GetBonds()]
[True, True, True, True, True, True]
```

Automatic aromaticity perception is normally the right thing to do, because different toolkits and even different versions of the same toolkit may have different ideas of what is aromatic, and it's best to ensure that they are consistently interpreted.

Aromaticity perception isn't needed when you know that the input aromaticity is correct and unambiguous. My timings show that aromaticity perception takes about half of the time needed to parse a SMILES string. If the string comes from a good data source, like a database record where OEChem created the SMILES, then you can nearly double the performance by omitting the perception step.

What does "ambiguous" mean? Consider azulene, which can be described by the SMILES "c1ccc2cccc2cc1". The fusion bond is not aromatic, while the peripheral bonds form a 10 pi electron system. In SMILES, an unspecified bond means "single or aromatic". If one of the terminal atoms is aliphatic then the bond must be a single bond. But as the fusion bond in azulene shows, it's possible for an unspecified bond with terminal aromatic atoms to still be non-aromatic. The above SMILES is ambiguous, and OEChem needs to do a full aromaticity analysis to determine that the fusion bond is not aromatic.

An unambiguous SMILES for azulene is "c1ccc-2cccc2cc1", where the fusion bond is marked explicitly as a single bond. The SMILES parser can use the simpler rule that an unspecified ring bond is aromatic whenever both terminal atoms are aromatic, and not require the lengthy aromatic perception step to determine that. OEChem generates unambiguous SMILES, so if you know OEChem generated the SMILES then you can recover the original aromaticity directly.

(As a side note, Daylight first introduced this in 4.71, and used fluorene ("C1c2ccccc2-c3ccccc13") as the prototypical case. Daylight's rule is to include the "-" for a single bond between two aromatic atoms, while OEChem's rule is to include the "-" for a single bond between two aromatic atoms and which is in a ring. Ring identification is much easier than aromaticity perception.)

So where was I ... ah, right, specifing the aromaticity model. I decided to separate aromaticity from the rest of the flavor flags, and specify it with its own *reader_args* and *writer_args* field. It's easiest to see using beneze in Kekule form:

```
>>> mol = openeye_toolkit.parse_molecule("C1=CC=CC=C1", "smistring",
...     reader_args={"aromaticity": "none"})
>>>
>>> [bond.IsAromatic() for bond in mol.GetBonds()]
[False, False, False, False, False, False]
>>> openeye_toolkit.create_string(mol, "smistring",
...     writer_args={"aromaticity": "none"})
'C1=CC=CC=C1'
```

NOTE: the aromaticity flags are volatile. If you don't specify the "none" aromaticity model then *chemfp.toolkit.create_string()* will reperceive aromaticity using the "openeye" aromaticity model and possibly reassign the aromaticity flags.

```
>>> openeye_toolkit.create_string(mol, "smistring")
u'c1ccccc1'
>>> openeye_toolkit.create_string(mol, "smistring",
...     writer_args={"aromaticity": "none"})
u'c1ccccc1'
```

This is consistent with how OEChem's high-level operations also modify the input molecule when creating output. I'm not fully happy with it. OEChem also has a "ConstMolecule" version, so this detail may change in the future.

Open Babel-specific SMILES reader_args and writer_args

In this section you'll learn how to pass toolkit-specific parameters to the Open Babel toolkit functions to create a SMILES string. You will need the Open Babel toolkit.

As far as I can tell, Open Babel does not have configuration options to change the default SMILES parser, so chemfp has no toolkit-specific *reader_args* for that toolkit. Open Babel does have configuration options to change the default SMILES output routines. These can be set in chemfp with the *writer_args* dictionary.

Open Babel uses an options string to change the configuration. The string "`i U smilesonly`" generates non-isomeric SMILES output, where the atom ordering is determined by the InChI's canonicalization algorithm ("Universal SMILES"), and where the identifier is excluded from the SMILES output.

Did you know all of that? I didn't. Some of these options are only documented in the code. It's also difficult for chemfp to handle since some of the options conflict with how chemfp thinks of things. For example, chemfp is in charge of including the identifier, so it will always enable "smilesonly", and it's difficult for the "cansmiles" output, which is non-isomeric, to know if an options string wants to override the default"i" option that it requires.

I ended up making my own *writer_args* API to have more explicit control over the individual parameters:

- explicit_hydrogens - boolean
- isomeric - boolean
- canonicalization - a string like "default", "none", "universal", "anticanonical", or "inchified"
- options - the Open Babel options string (if you must use it; using it may break things if you are not very careful.)

Here's an example of how to disable isomeric support for the "smistring" output, which would normally generate an isomeric SMILES:

```
>>> from chemfp import openbabel_toolkit
>>> mol = openbabel_toolkit.parse_molecule("[16O]=O", "smistring")
>>> openbabel_toolkit.create_string(mol, "smistring")
u'[16O]=O'
>>> openbabel_toolkit.create_string(mol, "smistring",
...     writer_args={"isomeric": False})
u'O=O'
```

I can also enable isomeric SMILES for the "canstring" format, which is normally non-isomeric:

```
>>> openbabel_toolkit.create_string(mol, "canstring")
u'O=O'
>>> openbabel_toolkit.create_string(mol, "canstring",
...     writer_args={"isomeric": True})
u'[16O]=O'
```

Open Babel supports several different canonicalization algorithms. Perhaps the most unusual one is "anticanonical", which uses random numbers for the atom ordering algorithm. The same molecule can generate different SMILES strings across multiple calls, so it's the antithesis of "canonical":

```
>>> for i in range(5):
...   print(openbabel_toolkit.create_string(mol, "smistring",
...         writer_args={"canonicalization": "anticanonical"}))
...
[16O]=O
[16O]=O
O=[16O]
[16O]=O
[16O]=O
```

See *Get the default reader_args or writer_args for a format* for a description of how to get the default reader and writer arguments for a given format, and use help(*openbabel_toolkit.read_molecules*) and help(*openbabel_toolkit.open_molecule_writer*) to get a more human-readable description.

Get the default reader_args or writer_args for a format

In this section you'll learn how to get the default *reader_args* and *writer_args* for a given format.

As you've seen, each toolkit format can have its own *reader_args* and *writer_args* parameters, and chemfp layers its own format types (like "smistring") on top of the native formats. It's easy to forget the specific parameters for a given format, much less the default values.

The *get_default_reader_args()* and *get_default_writer_args()* methods of the *Format* object return the respective default arguments:

```
>>> from chemfp import rdkit_toolkit
>>> fmt = rdkit_toolkit.get_format("smi")
>>> fmt.get_default_reader_args()
{'delimiter': None, 'has_header': False, 'sanitize': True}
>>> fmt.get_default_writer_args()
{'isomericSmiles': True, 'delimiter': None, 'kekuleSmiles': False, 'allBondsExplicit
↪': False, 'canonical': True}
```

You can sometimes use this information to see how chemfp maps its format types to the toolkit parameters. In RDKit, the difference between chemfp's "smi" and "can" formats is that *isomericSmiles* is True for the first and False for the second:

```
>>> rdkit_toolkit.get_format("can").get_default_writer_args()
{'isomericSmiles': False, 'delimiter': None, 'kekuleSmiles': False, 'allBondsExplicit
↪': False, 'canonical': True}
```

While writing this documentation I realized that the OEChem toolkit shows neither the default flavor nor the default aromaticity for a given format type. I will likely fix that in a future version of chemfp.

Convert text settings into reader and writer arguments

In this section you'll learn how to convert text-based configuration settings into the appropriate *reader_args* or *writer_args* dictionary.

The *reader_args* and *writer_args* take native Python values, including integers and booleans. In practice these will often be defined in a configuration file, through command-line options, or as CGI parameters. The *Format* methods *get_reader_args_from_text_settings()* and *get_writer_args_from_text_settings()* convert a text-based settings dictionary into the appropriate arguments dictionary with native Python objects as values. (These are methods of the Format object, because the parameter details are format-specific.)

The following shows an example using the RDKit toolkit's "sdf" format to get *reader_args* from a dictionary of text settings:

```
>>> from chemfp import rdkit_toolkit as T
>>>
>>> sdf_format = T.get_format("sdf")
>>> sdf_format.get_default_reader_args()
{'strictParsing': True, 'removeHs': True, 'sanitize': True}
>>>
>>> sdf_format.get_reader_args_from_text_settings({
...     "strictParsing": "true",
...     "removeHs": "False",
...     "sanitize": "0"})
{'strictParsing': True, 'removeHs': False, 'sanitize': False}
```

The boolean setting parser converts "true", "True", and "1" to Python's True, and "false", "False", and "0" to Python's False. Otherwise it raises a ValueError.

The following shows an equivalent example for RDKit's SDF *writer_args*:

```
>>> sdf_format.get_default_writer_args()
{'kekulize': True, 'includeStereo': False}
>>> sdf_format.get_writer_args_from_text_settings({
...     "kekulize": "false",
...     "includeStereo": "True"})
{'kekulize': False, 'includeStereo': True}
```

WARNING: these functions will ignore unknown keys. This was done to allow the text settings dictionary to contain settings for other toolkits and formats. As a result, typos are harder to detect, because they will be ignored.

See *argparse text settings to reader and writer args* for an example of converting text settings from the command-line into reader and writer arguments.

Multi-toolkit reader_args and writer_args

In this section you'll learn how to configure *reader_args* and *writer_args* so the same dictionary can be used to configure multiple toolkits and formats.

Sometimes you don't know which toolkit will be used for parsing, but you do know that you want Open Babel, OEChem, and RDKit to act in non-standard ways. For example, the choice of toolkit may depend on the user-defined fingerprint type, or simply (as in the following example) depend on user input.

The *reader_args* and *writer_args* will ignore unknown parameters, which lets you combine arguments for different toolkits into a single dictionary. As the toolkits use completely different parameter names (except a couple, like "delimiter", which are supposed to act the same for all toolkits), there's no conflict in the names for a given format.

The following defines a *reader_args* dictionary and a *writer_args* dictionary with parameters for each supported toolkit, then enters a loop. The loop asks the user for a SMILES string, or the name of the toolkit to use, or "q" to quit the loop. It will parse each SMILES into a molecule, then generate a SMILES output, although with decidedly strange parameters:

```
from __future__ import print_function
import chemfp
from chemfp import rdkit_toolkit as T  # use your default toolkit of choice

try:
  raw_input  # Python 2 name
except NameError:
  raw_input = input # Python 3

reader_args = {
   "sanitize": False,  # RDKit,
   "flavor": "Default|Strict",   # OEChem
   "aromaticity": "none",  # OEChem
}

writer_args = {
   "kekuleSmiles": True,  # RDKit
   "canonicalization": "anticanonical",  # Open Babel
   "aromaticity": "daylight",  # OEChem
}

print("Using", T.name, "toolkit")
while 1:
  query = raw_input("SMILES, toolkit name, or 'q' to quit? ")
  if not query or query == "q":
```

```
    break

  if query in ("rdkit", "openeye" ,"openbabel"):
    try:
        T = chemfp.get_toolkit(query)
    except ValueError:
        print("Toolkit %r not available" % (query,))
    print("Using", T.name, "toolkit")
    continue

  mol = T.parse_molecule(query, "smistring", reader_args=reader_args, errors="ignore")
  if mol is None:
    print("Toolkit", T.name, "could not parse query as SMILES")
    continue

  smiles = T.create_string(mol, "smistring", writer_args=writer_args, errors="ignore")
  if not smiles:
    print("Toolkit", T.name, "could not convert the molecule to SMILES")
    continue
  print(" -->", smiles)
```

I saved the above to a script and then ran it. It starts using RDKit, where I've set the reader's "sanitize" to False so RDKit won't perceive aromaticity on input, and set the writer's "kekuleSmiles" to show explicit aromatic bond types:

```
Using rdkit toolkit
SMILES, toolkit name, or 'q' to quit? C1=CC=CC=C1O
 --> OC1=CC=CC=C1
SMILES, toolkit name, or 'q' to quit? c1ccccc1O
 --> OC1:C:C:C:C:C:1
```

I then switch to the OpenEye toolkit, show that it is operating with "strict" added to the default reader flavor, and convert a couple of SMILES to canonical SMILES to show the output uses the Daylight aromaticity model instead of the default:

```
SMILES, toolkit name, or 'q' to quit? openeye
SMILES, toolkit name, or 'q' to quit? C==C
Warning: Problem parsing SMILES:
Warning: Bond without end atom.
Warning: C==C
Warning:   ^

Toolkit openeye could not parse query as SMILES
Using openeye toolkit
SMILES, toolkit name, or 'q' to quit? C1=CC=CC=C1O
 --> c1ccc(cc1)O
SMILES, toolkit name, or 'q' to quit? c1ccccc1O
 --> c1ccc(cc1)O
```

Finally, I switched to the Open Babel toolkit and showed that it generates "anti-canonical" SMILES, where the spanning tree priority order for SMILES output is randomly assigned:

```
SMILES, toolkit name, or 'q' to quit? openbabel
Using openbabel toolkit
SMILES, toolkit name, or 'q' to quit? C1=CC=CC=C1O
 --> Oc1ccccc1
SMILES, toolkit name, or 'q' to quit? C1=CC=CC=C1O
 --> Oc1ccccc1
SMILES, toolkit name, or 'q' to quit? C1=CC=CC=C1O
```

```
 --> c1ccc(cc1)O
SMILES, toolkit name, or 'q' to quit? c1ccccc1O
 --> Oc1ccccc1
SMILES, toolkit name, or 'q' to quit? c1ccccc1O
 --> c1c(O)cccc1
SMILES, toolkit name, or 'q' to quit? q
```

See *argparse text settings to reader and writer args* for an example of using multi-toolkit reader_args and writer_args.

Qualified reader and writer parameters names

In this section you'll learn how to use qualified parameter names. These give fine-grained control over the configuration options for each toolkit and format.

The previous section pointed out that the three toolkits use different parameter names, so for a given format you can combine the toolkit-specific *reader_args* into one unified dictionary and *writer_args* into another unified dictionary. However, within a toolkit the same parameter name can be reused for different formats, with different meanings.

This best example is for the *chemfp.openeye_toolkit*, where the *reader_args* and *writer_args* for all formats support the "flavor" and "aromaticity" parameters. The following shows examples where I might use a different flavor for the SMILES and InChI outputs, to get something other than the default representation:

```
>>> from chemfp import openeye_toolkit
>>> mol = openeye_toolkit.parse_molecule("CC([O-])=O", "smistring")
>>>
>>> openeye_toolkit.create_string(mol, "smistring")
u'CC(=O)[O-]'
>>> openeye_toolkit.create_string(mol, "smistring",
...     writer_args={"flavor": "Default|ImpHCount"})
u'[CH3]C(=O)[O-]'
>>>
>>> openeye_toolkit.create_string(mol, "inchistring")
u'InChI=1S/C2H4O2/c1-2(3)4/h1H3,(H,3,4)/p-1'
>>> openeye_toolkit.create_string(mol, "inchistring",
...     writer_args={"flavor": "Default|FixedHLayer"})
u'InChI=1/C2H4O2/c1-2(3)4/h1H3,(H,3,4)/p-1/fC2H3O2/q-1'
```

Chemfp uses "qualified" parameter names to handle this situation. For example, the qualified name "smistring.flavor" is the flavor parameter for the smistring format:

```
>>> writer_args = {
...    "smistring.flavor": "Default|ImpHCount",
...    "inchistring.flavor": "Default|FixedHLayer",
... }
>>> mol = openeye_toolkit.parse_molecule("CC([O-])=O", "smistring")
>>> openeye_toolkit.create_string(mol, "smistring", writer_args=writer_args)
u'[CH3]C(=O)[O-]'
>>> openeye_toolkit.create_string(mol, "inchistring", writer_args=writer_args)
'InChI=1/C2H4O2/c1-2(3)4/h1H3,(H,3,4)/p-1/fC2H3O2/q-1'
```

WARNING: there are six SMILES-related formats ("smi", "can", "usm", "smistring", "canstring", and "usmstring") so to be complete you'll need to specify values for all of them. There are also two InChI-related formats ("inchi" and "inchistring").

A "fully qualified" name looks like "openeye.smistring.flavor". The first term is the toolkit, the second the format name, and the last the parameter name. At present there is no real need for fully qualified names because the toolkits don't share any parameter names except for a couple which are supposed to be identical across all toolkits.

The following demonstration, which is more a parlor trick than something useful, shows how to have each toolkit use a different SMILES delimiter:

```
>>> from __future__ import print_function
>>> import chemfp
>>>
>>> reader_args = {
...   "rdkit.smi.delimiter": "tab",
...   "openbabel.smi.delimiter": "whitespace",
...   "openeye.smi.delimiter": "to-eol",
... }
>>>
>>> for toolkit_name in ("rdkit", "openbabel", "openeye"):
...   T = chemfp.get_toolkit(toolkit_name)
...   id, mol = T.parse_id_and_molecule("C\tabc def\tghi", "smi",
...                                     reader_args=reader_args)
...   print(toolkit_name, "sees the id", repr(id))
...
rdkit sees the id u'abc def'
openbabel sees the id u'abc'
openeye sees the id u'abc def\tghi'
```

(As a reminder, the 'delimiter' implementation is not perfect. A toolkit may accept the first whitespace after the SMILES term as a valid delimiter even if it doesn't match the actual parameter, and a toolkit may decide to stop parsing the SMILES term at the first whitespace.)

The final type of qualified parameter looks like "openeye.*.aromaticity", where the first term is the toolkit name, the second term is "*", and the third term is the parameter name. This is most useful if you want OEChem to enforce the same aromaticity across all formats, or have the RDKit parsers ignore sanitization, with configuration entries like:

```
{"openeye.*.aromaticity": "daylight",
 "rdkit.*.sanitize": False}
```

However, as only OEChem supports "aromaticity" and only RDKit supports "sanitize", you could also write this as simply:

```
{"aromaticity": "daylight",
 "sanitize": False}
```

The reason qualifier exist, even if not currently needed, is because I predict there will be parameter name conflicts in the future. That possibility affects the API enough that I wanted a solution now.

Qualified parameter priorities

In this section you'll learn the priority order when multiple terms try to specify the same parameter.

In the previous section you learned how "delimiter", "smi.delimiter", "rdkit.*.delimiter" and "rdkit.smi.delimiter" can all be used to set the delimiter style for RDKit's "smi" format. If more then one term is specified, which one wins?

Chemfp checks for the parameters in the following order:

1. rdkit.smi.delimiter
2. rdkit.*.delimiter
3. smi.delimiter
4. delimiter

The parameter with the highest ranking determines the setting, as the following shows:

```
>>> from chemfp import rdkit_toolkit as T
>>> id, mol = T.parse_id_and_molecule("C methane 16.04246", "smi",
...     reader_args={"delimiter": "to-eol",
...                  "smi.delimiter": "whitespace"})
>>> id
u'methane'
>>> id, mol = T.parse_id_and_molecule("C methane 16.04246", "smi",
...     reader_args={"rdkit.*.delimiter": "to-eol",
...                  "smi.delimiter": "whitespace"})
>>> id
u'methane 16.04246'
>>> id, mol = T.parse_id_and_molecule("C methane 16.04246", "smi",
...     reader_args={"rdkit.*.delimiter": "to-eol",
...                  "rdkit.smi.delimiter": "whitespace"})
>>> id
u'methane'
```

One way to remember it is the longest name has priority.

It can be confusing to have a large dictionary with multiple format and toolkit qualifiers. The *get_unqualified_reader_args()* and *get_unqualified_writer_args()* methods of *Format* object will return the fully unqualified *reader_args* and *writer_args* for that format:

```
>>> fmt = T.get_format("smi")
>>> fmt.get_unqualified_reader_args({
...       "delimiter": "to-eol",
...       "smi.delimiter": "whitespace",
...     })
{'delimiter': 'whitespace', 'has_header': False, 'sanitize': True}
>>> fmt.get_unqualified_writer_args({
...       "delimiter": "space",
...       "smi.delimiter": "tab",
...     })
{'isomericSmiles': True, 'delimiter': 'tab', 'kekuleSmiles': False, 'allBondsExplicit
↪': False, 'canonical': True}
```

This can also be helpful if you think you made a typo; get the unqualified *reader_args* and see if the result has the arguments you think it should have.

Qualified names and text settings

In this section you'll learn how the qualified names also apply to text settings.

Earlier you learned that text settings are string-based keys and values, which might come from the command-line, a configuration file, or some other text-based source. These need to be converted into Python values before they can be used as reader_args or writer_args.

A *Format* object can convert a dictionary of text settings into the correct argument dictionary. To get a Format object, ask the toolkit for the format of the given name:

```
>>> from chemfp import rdkit_toolkit as T
>>> fmt = T.get_format("sdf")
>>> fmt.get_default_reader_args()
{'strictParsing': True, 'removeHs': True, 'sanitize': True}
```

The section *Convert text settings into reader and writer arguments* showed how to convert the text settings with unqualified names into a *reader_args* dictionary:

```
>>> fmt.get_reader_args_from_text_settings({
...     "strictParsing": "false",
...     "removeHs": "false",
...     })
{'strictParsing': False, 'removeHs': False}
```

The text settings dictionary also supports qualified parameter names, including handling the priority resolution described in *Qualified parameter priorities*:

```
>>> fmt.get_reader_args_from_text_settings({
...     "strictParsing": "false",
...     "sdf.strictParsing": "true",
...     "removeHs": "false",
...     "rdkit.*.removeHs": "true",
...     "rdkit.sdf.sanitize": "false",
...     })
{'strictParsing': True, 'removeHs': True, 'sanitize': False}
```

If you stare at it for a bit you'll see that "sdf.strictParsing" has a higher priority than "strictParsing" and "rdkit.*.removeHs" is higher than "removeHs", which is how it's supposed to work.

Read molecules from an SD file or stdin

In this section you'll learn how to read an SD file and iterate through its records as toolkit molecules. You will need Compound_027575001_027600000.sdf.gz from PubChem.

Time to get back to molecules! The *chemfp.toolkit.read_molecules()* function reads molecules from a structure file:

```
from __future__ import print_function
from chemfp import rdkit_toolkit as T  # use your toolkit of choice
for mol in T.read_molecules("Compound_014550001_014575000.sdf.gz"):
    print(T.create_string(mol, "smistring"))
```

By default it uses the filename extension to figure out the format and compression type. You can specify it yourself, if you wish, using the *format* option:

```
from __future__ import print_function
from chemfp import rdkit_toolkit as T  # use your toolkit of choice
for mol in T.read_molecules("Compound_014550001_014575000.sdf.gz",
                            format="sdf.gz"):
  print(T.create_string(mol, "smistring"))
```

Examples of valid format values are "smi", "can", and "usm" (but not the *string variants like "smistring", because those aren't record-based formats), and "sdf", as well as gzip-compressed versions like "smi.gz" and "sdf.gz".

(For Open Babel the ".gz" extension does nothing as Open Babel will auto-detect and handle gzip compressed input.)

If the first parameter (the *source* parameter) is the Python None value then the toolkit will read from stdin. As there's no filename, chemfp can't look at the extension to figure out the format, so it assumes the input is in "smi" format, that is, an uncompressed SMILES file.

Therefore, to read an SD file from stdin you must specify the format. The following program reads a gzip compressed SD file from stdin, convert it to SMILES, and find the 10 most common characters used in the SMILES strings:

```
# This file is named 'count_smiles_characters.py'
from __future__ import print_function
from collections import Counter
from chemfp import rdkit_toolkit as T # use your toolkit of choice

symbol_counts = Counter()
for mol in T.read_molecules(None, "sdf.gz"):
  smiles = T.create_string(mol, "smistring")
  symbol_counts.update(smiles)
for symbol, count in symbol_counts.most_common(10):
  print("%5d: %r" % (count, symbol))
```

Now to try it on a data set:

```
% python count_smiles_characters.py < Compound_014550001_014575000.sdf.gz
50826: 'C'
38147: 'c'
21010: '('
21010: ')'
15316: 'O'
12676: '1'
 9054: '='
 7444: '2'
 5755: '@'
 5427: '['
```

(I double-checked; the next most common is indeed '[' with 5427 occurrences, which you probably expected. :)

Read ids and molecules from an SD file at the same time

In this section you'll learn how to read an SD file and iterate through its records as the two-element tuple of (id, molecule). You will need the Compound_027575001_027600000.sdf.gz from PubChem, which was used in the previous section.

In an earlier section, *Parse the id and the molecule at the same time*, you learned how to parse a structure record to get both the identifier and the molecule at the same time. The toolkit function *chemfp.toolkit.read_ids_and_molecules()* is the equivalent for reading from a structure file.

In the following example I'll use the RDKit toolkit to create a tab-separated file with the id in the first column, the number of carbon atoms in the second, and the SMILES in the third. For brevity, I'll display only the first 10 records, which also gives a nice example of when to use itertools.islice:

```
from __future__ import print_function
from itertools import islice
from chemfp import rdkit_toolkit
filename = "Compound_014550001_014575000.sdf.gz"
reader = rdkit_toolkit.read_ids_and_molecules(filename)

for id, mol in islice(reader, 0, 10):
    num_carbons = sum(1 for atom in mol.GetAtoms() if atom.GetAtomicNum() == 6)
    smiles = rdkit_toolkit.create_string(mol, "smistring")
    print(id, num_carbons, smiles, sep="\t")
```

(See the next section for a description of how the line with the `sum()` works.)

Here's the output, and a spot check shows the carbon counts are correct:

```
14550001       9       O=[N+]([O-])c1ccc(O)c(CSCCO)c1
14550002       9       Nc1ccc(O)c(CSCCO)c1
14550003       8       CSCc1cc(N)ccc1O
14550004       7       Cc1[nH]ncc1C(C)C
14550005       12      O=C([O-])c1ccc2cc(C(=O)[O-])ccc2c1.[K+].[K+]
14550010       10      O=C(O)CN(Cc1ccccc1)CP(=O)(O)O
14550011       10      CCO[Si]1(C)OC(=O)c2ccccc2O1
14550044       54      CCCCCCCCCCCCCCCCCC(=O)[O-].CCCCCCCCCCCCCCCCCC(=O)[O-].
↪CCCCCCCCCCCCCCCCCC(=O)[O-].[Gd+3]
14550045       19
↪O=C(O)CCCCCCCCCC(F)(F)C(F)(F)C(F)(F)C(F)(F)C(F)(F)C(F)(F)C(F)(F)C(F)(F)C(F)(F)F
14550054       14      c1ccc(SCSCSc2ccccc2)cc1
```

What's fun is that RDKit and OEChem both implement `mol.GetAtoms()` and `atom.GetAtomicNum()` so to port the above from RDKit to OEChem is trivial; replace `rdkit_toolkit` with `openeye_toolkit`!

The Open Babel port isn't quite as easy because Open Babel has a different way to get the atoms in a molecule. To make it easy to copy and paste, here's the equivalent code for Open Babel:

```
from __future__ import print_function
from itertools import islice
from chemfp import openbabel_toolkit
filename = "Compound_014550001_014575000.sdf.gz"
reader = openbabel_toolkit.read_ids_and_molecules(filename)

for id, mol in islice(reader, 0, 10):
  num_carbons = sum(1 for atom_idx in range(mol.NumAtoms())
                          if mol.GetAtom(atom_idx+1).GetAtomicNum() == 6)
  smiles = openbabel_toolkit.create_string(mol, "smistring")
  print(id, num_carbons, smiles, sep="\t")
```

Read ids and molecules using an SD tag for the id

In this section you'll learn how to use the *id_tag* to get the id from one of the SD tags, rather than from the record's title. You will need the Compound_027575001_027600000.sdf.gz from PubChem, which was used in the previous section. I'll also explain an idiom for how to count the number of records in an iterator.

Sometimes you would rather use a tag value as the id rather than the title line of the SDF record. This is critical for ChEBI data set and older ChEMBL data sets, which leave the title line (mostly) blank. In this case, use the *id_tag* to specify the tag to use.

The following example modifies the RDKit code from previous code to use PUBCHEM_IUPAC_SYSTEMATIC_NAME as the id, rather than the title line:

```
from __future__ import print_function
from itertools import islice
from chemfp import rdkit_toolkit
filename = "Compound_014550001_014575000.sdf.gz"
reader = rdkit_toolkit.read_ids_and_molecules(filename, id_tag="PUBCHEM_IUPAC_
↪SYSTEMATIC_NAME")

for id, mol in islice(reader, 0, 10):
  num_carbons = sum(1 for atom in mol.GetAtoms() if atom.GetAtomicNum() == 6)
  smiles = rdkit_toolkit.create_string(mol, "smistring")
  print(id, num_carbons, smiles, sep="\t")
```

The first 7 lines of output is:

```
2-(2-hydroxyethylsulfanylmethyl)-4-nitro-phenol        9       O=[N+]([O-
↪])c1ccc(O)c(CSCCO)c1
4-azanyl-2-(2-hydroxyethylsulfanylmethyl)phenol        9       Nc1ccc(O)c(CSCCO)c1
4-azanyl-2-(methylsulfanylmethyl)phenol       8       CSCc1cc(N)ccc1O
5-methyl-4-propan-2-yl-1H-pyrazole    7       Cc1[nH]ncc1C(C)C
dipotassium;naphthalene-2,6-dicarboxylate     12      O=C([O-])c1ccc2cc(C(=O)[O-
↪])ccc2c1.[K+].[K+]
2-[(phenylmethyl)-(phosphonomethyl)amino]ethanoic acid        10      ␣
↪O=C(O)CN(Cc1ccccc1)CP(=O)(O)O
2-ethoxy-2-methyl-1,3,2-benzodioxasilin-4-one 10      CCO[Si]1(C)OC(=O)c2ccccc2O1
```

You might have found the "`sum(1 for atom in ....)`" a bit odd. I agree with you. It is, however, the standard way in Python to count the number of elements in the iterator which match a given condition. I'll break it down so you can understand how it works.

A list comprehension iterates through each element in an iterator (in the following it iterates over the characters in a string) and returns a list:

```
>>> [c for c in "Hello"]
['H', 'e', 'l', 'l', 'o']
```

Add an "if" to it to operate on only a subset of the characters:

```
>>> [c for c in "Hello" if c != "l"]
['H', 'e', 'o']
```

I could use len() of this to get the number of non-"l" characters, but that would require making a list only to throw it away. There's another route to the same answer. To get there, use the value 1 for each character rather than the character itself:

```
>>> [1 for c in "Hello" if c != "l"]
[1, 1, 1]
```

Then use sum() to sum the values, which in this case is also the number of elements in the list:

```
>>> sum([1 for c in "Hello" if c != "l"])
3
```

Unlike len(), sum() only needs an iterator, not a list. I can replace the list comprehension with a generator comprehension, to get:

```
>>> sum(1 for c in "Hello" if c != "l")
3
```

Going back to the RDKit/OEChem expression:

```
num_carbons = sum(1 for atom in mol.GetAtoms() if atom.GetAtomicNum() == 6)
```

I hope you can see how this counts the number of atoms in the molecule whose atomic number is 6. Or, if you want another way to think of it, the expression is the same as:

```
num_carbons = 0
for atom in mol.GetAtoms():
    if atom.GetAtomicNum() == 6:
        num_carbons += 1
```

Read from a string instead of a file

In this section you'll learn how to read molecules from a string containing multiple SMILES records.

In the section *Read molecules from an SD file or stdin* you learned how to read molecules from a structure file or stdin. Sometimes the input structures come from a string. For example, if a web page has a form with a text box, where users can paste in a set of SMILES or SDF records and submit the form, then the web application on the server will likely receive those records as a single string.

When the records are in a string instead of a file, use *chemfp.toolkit.read_molecules_from_string()*. It's very similar to *chemfp.toolkit.read_molecules()*, except that the first parameter, *content*, is the string instead of the source filename, and the second parameter, *format*, is required. (chemfp doesn't try to auto-detect the format from the content.)

The following reads the records from a string containing two simple SMILES records and prints the number of non-implicit atoms for each one. I've included implementations for all three toolkits; use the one(s) that are available to you:

```
from __future__ import print_function
from chemfp import rdkit_toolkit
content = ("C methane 16.04246\n"
           "O=O water 31.9988\n")

for mol in rdkit_toolkit.read_molecules_from_string(content, "smi"):
  print("RDKit:", mol.GetNumAtoms())

from chemfp import openeye_toolkit
for mol in openeye_toolkit.read_molecules_from_string(content, "smi"):
  print("OEChem:", mol.NumAtoms())

from chemfp import openbabel_toolkit
for mol in openbabel_toolkit.read_molecules_from_string(content, "smi"):
  print("Open Babel:", mol.NumAtoms())
```

When I run the above (on a computer where all three supported toolkits are installed), the above reports:

```
RDKit: 1
RDKit: 2
OEChem: 1
OEChem: 2
Open Babel: 1
Open Babel: 2
```

I would like to improve the output a bit to also include the record id in the output. The toolkit function *chemfp.toolkit.read_ids_and_molecules_from_string()* is similar to *chemfp.toolkit.read_molecules_from_string()* except that it iterates through the (id, toolkit molecule) tuple rather than just the molecule:

```
>>> from __future__ import print_function
>>> from chemfp import rdkit_toolkit
>>> content = ("C methane 16.04246\n"
...            "O=O water 31.9988\n")
>>> for (id, mol) in rdkit_toolkit.read_ids_and_molecules_from_string(content, "smi"):
...     print("RDKit:", repr(id), mol.GetNumAtoms())
...
RDKit: 'methane 16.04246' 1
RDKit: 'water 31.9988' 2
```

You can see that the default SMILES reader assumes the rest of the line is the id. The file and string record readers take a *reader_args* parameter just like *chemfp.toolkit.parse_id_and_molecule()*. I'll specify the "whitespace" delimiter so the parser uses only the second word as the id:

```
>>> for (id, mol) in rdkit_toolkit.read_ids_and_molecules_from_string(content, "smi",
...                       reader_args={"delimiter": "whitespace"}):
...     print("RDKit:", repr(id), mol.GetNumAtoms())
...
RDKit: 'methane' 1
RDKit: 'water' 2
```

See *Specify a SMILES delimiter through reader_args* for more details about setting the "delimiter" *reader_args*.

The string readers, like the file readers, also support the *id_tag* option to get the id from an SD tag instead of the title line. See *Read ids and molecules using an SD tag for the id* for more details about using the *id_tag*.

The reader may reuse molecule objects!

In this section you'll learn that the OEChem and Open Babel toolkits reuse the same molecule object, which means you can't save a molecule for later.

Suppose you want to read all of the molecules from a file into a list. It's very tempting to write it as:

```
>>> import chemfp
>>> from chemfp import openeye_toolkit as T
>>> mols = list(T.read_molecules_from_string("C methane\nO water\n", "smi"))
```

This does not work for the openeye_toolkit or the openbabel_toolkit:

```
>>> mols
[<openeye.oechem.OEGraphMol; proxy of <Swig Object of type 'OEGraphMolWrapper *' at␣
↪0x10326ba40> >,
 <openeye.oechem.OEGraphMol; proxy of <Swig Object of type 'OEGraphMolWrapper *' at␣
↪0x10326ba40> >]
>>> T.create_string(mols[0], "smistring")
u''
>>> T.create_string(mols[1], "smistring")
u''
```

This is because the underlying reader for those two toolkits reuse the same molecule object. You can see that in the above, which returns the same OEGraphMol object (with id 0x10326ba40) for each record. The reason why Open Eye decided to reuse the object is to get better performance. Clearing the molecule object is faster than deleting it and reallocating a new one.

In addition, the OEChem reader code does a "clear molecule" followed by "read next record or stop". At the end of the file there is no record, so the reader ends with a clear molecule. That explains why the OEGraphMol produces an empty SMILES string for the last couple of lines in the above code.

The only portable way to load a list of molecules is to use *chemfp.toolkit.copy_molecule()*, as in:

```
>>> from chemfp import openeye_toolkit as T
>>> mols = [T.copy_molecule(mol) for mol in T.read_molecules_from_string("C␣
↪methane\nO water\n", "smi")]
>>> mols
[<openeye.oechem.OEGraphMol; proxy of <Swig Object of type 'OEGraphMolWrapper *' at␣
↪0x10328a810> >,
 <openeye.oechem.OEGraphMol; proxy of <Swig Object of type 'OEGraphMolWrapper *' at␣
↪0x100c78320> >]
```

```
>>> T.create_string(mols[0], "smistring")
u'C'
>>> T.create_string(mols[1], "smistring")
u'O'
```

I don't really like this solution because the RDKit reader doesn't need a copy, so the extra copy is pure overhead.

Future versions of chemfp will likely have a *reader_arg* to specify if it's okay to reuse a molecule object or if a new one must be used each time.

Write molecules to a SMILES file

In this section you will learn how to write toolkit molecules into a structure file. You will need Compound_014550001_014575000.sdf.gz from PubChem, which is a different PubChem file than what I used in earlier sections.

Chemfp can write toolkit molecules to a file in a given format. I'll start by making an RDKit molecule, though the same API works with Open Babel and OEChem:

```
>>> from chemfp import rdkit_toolkit as T  # use your toolkit of choice
>>> mol = T.parse_molecule("c1ccccc1O phenol", "smi")
```

Use *chemfp.toolkit.open_molecule_writer()* to create a writer object. By default it will look at the output filename extension to figure out the format and compression type, and if that doesn't work it defaults to SMILES output:

```
>>> writer = T.open_molecule_writer("example.smi")
```

The fingerprint writer has several methods to write a molecule to the file. If you write a molecule by itself it will use the molecule's own id (in this case, "phenol"):

```
>>> writer.write_molecule(mol)
```

Or, use *write_id_and_molecule()* if you want to specify an alternate id:

```
>>> writer.write_id_and_molecule("something else", mol)
```

WARNING: The toolkit implementation may temporarily change the toolkit molecule's own identifier in order to get the correct output. You should not alter the molecule's id in another thread while calling this function.

Let's see if it worked, by closing the writer (otherwise some of the output may be in an internal buffer) and reading the file:

```
>>> writer.close()
>>> print(open("example.smi").read())
Oc1ccccc1 phenol
Oc1ccccc1 something else
```

The *write_molecules()* method is optimized for passing in a list or iterator of molecule objects, and *write_ids_and_molecules()* is the equivalent if you have (id, molecule) pairs. For example, the following converts an SD file into a compressed SMILES file:

```
from chemfp import rdkit_toolkit as T # use your toolkit of choice
reader = T.read_molecules("Compound_014550001_014575000.sdf.gz")
writer = T.open_molecule_writer("example.smi.gz")
writer.write_molecules(reader)
```

```
# These are optional, but recommended. Even better would be
# to use the context manager described in the next section.
writer.close()
reader.close()
```

If you have a list (or iterator) of molecules, then use the *write_molecules()* method.

The open function also supports the *format* parameter, so you can specify "smi" or "sdf.gz" some other combination of structure format and compression type:

```
writer = T.open_molecule("wrong_extensions.smi", format="sdf.gz")
```

Reader and writer context managers

In this section you'll learn how to use chemfp's readers and writers to close the file, rather than depend on Python's garbage collector or manual "close()". You will need Compound_014550001_014575000.sdf.gz from PubChem.

In the previous section, *Write molecules to a SMILES file*, you learned how to convert an SD file into a SMILES file. At the end was a small program with optional "close()" statements. These are optional because Python's garbage collector and chemfp work together. When a chemfp reader or writer is no longer needed, the garbage collector asks chemfp to clean up, and chemfp closes the native toolkit's file object.

This is fine for a simple script or function, but sometimes you want more control over when the file is closed. You can call the writer's *close()* method yourself, but it's really easy to forget to do that.

Python supports "context managers", which carry out certain actions when a block of code finishes. See **PEP 343** if you want the full details. For chemfp you only need to know that the reader and writer context managers will always close the file at the end of the block.

A normal Python file context manager works like this:

```
>>> with open("example.txt", "w") as outfile:
...     outfile.write("I am here.\n")
...
>>> print(repr(open("example.txt").read()))
'I am here.\n'
```

If instead I use one file object to write the data and another to read the file, without a flush() or close() by the writer, then there's a syncronization problem:

```
>>> outfile = open("example.txt", "w")
>>> outfile.write("I am here.\n")
>>> print(repr(open("example.txt").read()))
''
```

Why does this print the empty string? The output text is still in an internal buffer, which isn't written to the disk until the close call:

```
>>> outfile.close()
>>> print(repr(open("example.txt").read())
'I am here.\n'
```

The same problem occurs with molecule output:

```
>>> from chemfp import rdkit_toolkit as T # can also use openbabel_toolkit
>>> mol = T.parse_molecule("C=O carbon monoxide", "smi")
```

```
>>> writer = T.open_molecule_writer("example.smi")
>>> writer.write_molecule(mol)
>>> open("example.smi").read()
''
>>> writer.close()
>>> open("example.smi").read()
'C=O carbon monoxide\n'
```

Note: this problem does not occur with the openeye_toolkit. Most likely that toolkit always flushes its output buffers after each molecule.

The chemfp readers and writers support a context manager, so you can use the same solution you would for regular files:

```
>>> from chemfp import rdkit_toolkit as T # use your toolkit of choice
>>> mol = T.parse_molecule("C=O carbon monoxide", "smi")
>>> with T.open_molecule_writer("example.smi") as writer:
...     writer.write_molecule(mol)
...
>>> open("example.smi").read()
'C=O carbon monoxide\n'
```

With the context manager concept firmly in mind, the following is the way I prefer to write the conversion script from the previous section:

```
from chemfp import rdkit_toolkit as T # use your toolkit of choice

with T.read_molecules("Compound_014550001_014575000.sdf.gz") as reader:
  with T.open_molecule_writer("example.smi.gz") as writer:
    writer.write_molecules(reader)
```

That said, if you really want to depend on the garbage collector, you can also write it with one (or two) fewer lines:

```
from chemfp import rdkit_toolkit as T # use your toolkit of choice
T.open_molecule_writer("example.smi.gz").write_molecules(
    T.read_molecules("Compound_014550001_014575000.sdf.gz"))
```

Write molecules to stdout in a specified format

In this section you'll learn how to specify the structure writer's output format, and to write to stdout instead of to a file.

The function *chemfp.toolkit.open_molecule_writer()* supports a *format* parameter, in case you don't want chemfp to determine the output format and compression based on the filename extension.

For example, if the destination is None (instead of a filename) then chemfp will write the output to stdout. Since Python's None object doesn't have an extension, it will write the molecules as uncompressed SMILES. If you want to write to stdout in SDF format you will have to specify the output format, like the following:

```
>>> from chemfp import rdkit_toolkit as T # use your toolkit of choice
>>> mol = T.parse_molecule("O=O molecular oxygen", "smi")
>>> with T.open_molecule_writer(None, "sdf") as writer:
...     writer.write_molecule(mol)
...
molecular oxygen
     RDKit

  2  1  0  0  0  0  0  0  0  0999 V2000
```

```
    0.0000    0.0000    0.0000 O   0  0  0  0  0  0  0  0  0  0  0  0
    0.0000    0.0000    0.0000 O   0  0  0  0  0  0  0  0  0  0  0  0
  1  2  2  0
M  END
$$$$
>>> with T.open_molecule_writer(None, "inchikey") as writer:
...   writer.write_molecule(mol)
...
MYMOFIZGZYHOMD-UHFFFAOYSA-N molecular oxygen
```

Write molecules to a string (and a bit of InChI)

In this section you'll learn how to write toolkit molecules into memory, and when finished to get the result as a string.

The previous sections showed examples of writing molecules to a file or to stdout. Sometimes you want to save the records as a string; perhaps to send a response for a web request or display the contents in a text pane of a GUI. The function *chemfp.toolkit.open_molecule_writer_to_string()* creates a *MoleculeStringWriter* which stores the output records into memory. Once the writer is closed, the memory contents can be retrieved as a string with *MoleculeStringWriter.getvalue()*.

For a bit of variation, the following example uses the "inchi" output format, and the *openbabel_toolkit*:

```
>>> from chemfp import openbabel_toolkit as T # use your toolkit of choice
>>> alanine = T.parse_molecule("O=C(O)[C@@H](N)C alanine", "smi")
>>> glycine = T.parse_molecule("C(C(=O)O)N glycine", "smi")
>>> writer = T.open_molecule_writer_to_string("inchi")
>>> writer.write_molecules([alanine, glycine])
>>> writer.close()
>>> print(writer.getvalue())
InChI=1S/C3H7NO2/c1-2(4)3(5)6/h2H,4H2,1H3,(H,5,6)/t2-/m0/s1 alanine
InChI=1S/C2H5NO2/c3-1-2(4)5/h1,3H2,(H,4,5) glycine
```

You should know that there's no well-defined "inchi" file format, only an InChI string. I decided to follow Open Babel's lead and say that the "inchi" format has one record per line, where each line contains the InChI string followed by a delimiter, followed by the id (if available) on the rest of the line.

The InChI output *writer_args* supports an "include_id" parameter. The default, True, includes the id, while the following example sets it to False to have only the InChI string on the line:

```
>>> with T.open_molecule_writer_to_string("inchi",
...                writer_args={"include_id": False}) as writer:
...   writer.write_molecule(alanine)
...   writer.write_molecule(glycine)
...
>>> print(writer.getvalue())
InChI=1S/C3H7NO2/c1-2(4)3(5)6/h2H,4H2,1H3,(H,5,6)/t2-/m0/s1
InChI=1S/C2H5NO2/c3-1-2(4)5/h1,3H2,(H,4,5)
```

I also used the context manager so the code would be a bit shorter and, I think, clearer. It's up to you to decide if `write_molecules()` with a 2-element list is clear than two `write_molecule()` lines.

Handling errors when reading molecules from a string

In this section you'll learn how to ignore errors and improve error reporting when reading from a string, rather then accept the default of raising an exception and stopping. The examples will use a string containing SMILES records,

but the same principles apply to any format.

If you've used the chemfp readers on real-world data sets you might have noticed that the RDKit and Open Babel ones sometimes raise an exception, saying that a given record could not be parsed. I'll demonstrate with a string containing four SMILES records:

```
>>> content = ("C methane\n" +
...            "CN(C)(C)(C)C pentavalent nitrogen\n" +
...            "Q Q-ane\n" +
...            "[U] uranium\n")
>>>
```

RDKit doesn't like the pentavalent nitrogen, and chemfp's rdkit_toolkit stops processing at that record:

```
>>> from chemfp import rdkit_toolkit
>>> with rdkit_toolkit.read_ids_and_molecules_from_string(content, "smi") as reader:
...     for id, mol in reader:
...         print(id)
...
methane
[16:11:12] Explicit valence for atom # 1 N, 5, is greater than permitted
Traceback (most recent call last):
  File "<stdin>", line 2, in <module>
  File "chemfp/_rdkit_toolkit.py", line 286, in _iter_read_smiles_ids_and_molecules
    error_handler.error("RDKit cannot parse the SMILES %s" % (_compat.myrepr(smiles),
↪), location)
  File "chemfp/io.py", line 87, in error
    _compat.raise_tb(ParseError(msg, location), None)
  File "<string>", line 1, in raise_tb
chemfp.ParseError: RDKit cannot parse the SMILES 'CN(C)(C)(C)C',
file '<string>', line 2, record #2: first line is 'CN(C)(C)(C)C pentavalent nitrogen'
```

Open Babel doesn't care about the too-high valence on the nitrogen, but doesn't like the non-SMILES in the third record:

```
>>> from chemfp import openbabel_toolkit
>>> with openbabel_toolkit.read_ids_and_molecules_from_string(content, "smi") as␣
↪reader:
...     for id, mol in reader:
...         print(id)
...
methane
pentavalent nitrogen
Traceback (most recent call last):
  File "<stdin>", line 2, in <module>
  File "chemfp/_openbabel_toolkit.py", line 894, in _iter_column_records
    % (format_name, myrepr(smi_string)), location)
  File "chemfp/io.py", line 87, in error
    _compat.raise_tb(ParseError(msg, location), None)
  File "<string>", line 1, in raise_tb
chemfp.ParseError: Open Babel cannot parse the SMILES 'Q', file '<string>', line 3,␣
↪record #3: first line is 'Q Q-ane'
```

To round things out, OEChem accepts pentavalent nitrogen and skips the bad SMILES at a lower level than what chemfp uses, so there's no exception:

```
>>> from chemfp import openeye_toolkit
>>> with openeye_toolkit.read_ids_and_molecules_from_string(content, "smi") as reader:
...     for id, mol in reader:
```

```
...         print(id)
...
methane
pentavalent nitrogen
Warning: Problem parsing SMILES:
Warning: Q Q-ane
Warning: ^

Warning: Error reading molecule "" in Canonical stereo SMILES format.
uranium
```

I'll emphasize that point. The openeye_toolkit uses OEChem's high-level reader, which provides no information about if OEChem skipped a record with a failure. Chemfp therefore cannot provide more information about the failures, whether as an exception or an improved error message.

I'm certain that nearly everyone wants the reader to ignore the few records that can't be parsed by the underlying toolkit. The readers and writers support the *errors* option. The default value of "strict" tells chemfp to raise an exception when it detects a parse failure, and "ignore" tells it to ignore the error and go on to the next record:

```
>>> with rdkit_toolkit.read_ids_and_molecules_from_string(
...                 content, "smi", errors="ignore") as reader:
...     for id, mol in reader:
...         print(id)
...
methane
[16:13:45] Explicit valence for atom # 1 N, 5, is greater than permitted
[16:13:45] SMILES Parse Error: syntax error for input: 'Q'
uranium
>>> with openbabel_toolkit.read_ids_and_molecules_from_string(
...                 content, "smi", errors="ignore") as reader:
...     for id, mol in reader:
...         print(id)
...
methane
pentavalent nitrogen
uranium
```

The "strict" default comes from my long-held belief that it's better to be strict first, and detect problems early, than to let them intrude. My resolve is weakening, because it's been rare to find that I can make use of that information. The biggest counter-example is when I specify one format but the file is actually in another format, in which case the reader skips a lot of garbage. For example, a SMILES reader, pointed to a SD file or a compressed SMILES file, will try hard to make sense of the data and end up ignoring almost everything. I haven't decided if I will change the default policy.

I've also found that the toolkits aren't that helpful at identifying which record failed. Take a look at the RDKit warning:

```
[16:13:45] Explicit valence for atom # 1 N, 5, is greater than permitted
```

It says that I did this in the late afternoon, and the reason for the failure, but says very little about the record with the problem.

To help improve this, and to send still more garbage, err, I mean helpful messages to stderr, chemfp supports a "report" *errors* value. It's the same as "ignore" except that it also displays more details about the failure location:

```
>>> with rdkit_toolkit.read_ids_and_molecules_from_string(
...                 content, "smi", errors="report") as reader:
...     for id, mol in reader:
...         print(id)
```

```
...
methane
[16:14:52] Explicit valence for atom # 1 N, 5, is greater than permitted
ERROR: RDKit cannot parse the SMILES 'CN(C)(C)(C)C', file '<string>', line 2, record
↪#2: first line is 'CN(C)(C)(C)C pentavalent nitrogen'. Skipping.
[16:14:52] SMILES Parse Error: syntax error for input: 'Q'
ERROR: RDKit cannot parse the SMILES 'Q', file '<string>', line 3, record #3: first␣
↪line is 'Q Q-ane'. Skipping.
uranium
>>> with openbabel_toolkit.read_ids_and_molecules_from_string(
...                content, "smi", errors="report") as reader:
...     for id, mol in reader:
...         print(id)
...
methane
pentavalent nitrogen
ERROR: Open Babel cannot parse the SMILES 'Q', file '<string>', line 3, record #3:␣
↪first line is 'Q Q-ane'. Skipping.
uranium
```

The quality of the error message depends on the toolkit and the format. The best messages are for the Open Babel and RDKit SMILES readers and InChI readers, because I decided to have chemfp identify the records for those formats itself, instead of using the underlying toolkits to read the file. Chemfp still uses the underlying toolkit to convert the individual record into a native toolkit molecule.

I did this because I found the the SMILES and InChI reader performance was the same, and by writing my own parsers I had the ability to report line numbers and improve the error messages.

The examples so far used the `read_ids_and_molecules_from_string` function. The `read_molecules_from_string` function also supports the `errors` option, with the same meaning.

```
>>> sizes = []
>>> with openbabel_toolkit.read_molecules_from_string(
...     content, "smi", errors="report") as reader:
...   for mol in reader:
...     sizes.append(mol.NumAtoms())
...
ERROR: Open Babel cannot parse the SMILES 'Q', file '<string>', line 3, record #3:␣
↪first line is 'Q Q-ane'. Skipping.
>>> sizes
[1, 6, 1]
```

Handling errors when reading molecules from a file

In this section you'll learn how to how to ignore errors and improve error reporting when reading from SD file, rather then accept the default of raising an exception and stopping. The examples will use an SD file, but the same principles apply to any format.

In the previous section you learned that when the readers encounter a error, the default behavior is to raise a Python exception and how to use the `error` parameter to ignore those errors or to provide a more detailed error report.

The file-based readers, *chemfp.toolkit.read_molecules()* and *chemfp.toolkit.read_ids_and_molecules()*, can be configured the same way, that is:

```
# When there is an error, raise an exception and stop (this is the default)
T.read_molecules(filename)
T.read_molecules(filename, errors="strict")
```

```
T.read_ids_and_molecules(filename)
T.read_ids_and_molecules(filename, errors="strict")

# When there is an error, go on to the next record
T.read_molecules(filename, errors="ignore")
T.read_ids_and_molecules(filename, errors="ignore")

# When there is an error, print an error message to stderr then
# go on to the next record
T.read_molecules(filename, errors="report")
T.read_ids_and_molecules(filename, errors="report")
```

To show it in action, I'll construct an SD file with three records. The first will contain a trivalent oxygen, the second a corrupt record, and the third will be atomic nitrogen. I'll use OEChem to help me make the file.

```
from chemfp import openeye_toolkit as T
mol1 = T.parse_molecule("O#C trivalent", "smi") # RDKit won't like this
mol2 = T.parse_molecule("[U] Q-record", "smi")  # I'll corrupt this record
mol3 = T.parse_molecule("[N] nitrogen", "smi")  # This one is fine
with T.open_molecule_writer_to_string("sdf") as writer:
  writer.write_molecules([mol1, mol2, mol3])
content = writer.getvalue()
# replace the "U" with the nonsense "Qq"
content = content.replace("U ", "Qq")
# Save
open("bad_data.sdf", "w").write(content)
```

Here's what the output file `bad_data.sdf` it looks like, so you can copy&paste if you wish:

```
trivalent
  -OEChem-04251716112D

  2  1  0     0  0  0  0  0  0999 V2000
    0.0000    0.0000    0.0000 O   0  0  0  0  0  0  0  0  0  0  0  0
    1.0000    0.0000    0.0000 C   0  0  0  0  0  0  0  0  0  0  0  0
  1  2  3  0  0  0  0
M  END
$$$$
Q-record
  -OEChem-04251716112D

  1  0  0     0  0  0  0  0  0999 V2000
    0.0000    0.0000    0.0000 Qq  0  0  0  0  0  0  0  0  0  0  0  0
M  END
$$$$
nitrogen
  -OEChem-04251716112D

  1  0  0     0  0  0  0  0  0999 V2000
    0.0000    0.0000    0.0000 N   0  0  0  0  0 15  0  0  0  0  0  0
M  END
$$$$
```

I'll try to read that file using the native RDKit reader, which skips records it can't parse:

```
>>> from rdkit import Chem
>>> reader = Chem.ForwardSDMolSupplier("bad_data.sdf")
>>> ids = [mol.GetProp("_Name") for mol in reader if mol is not None]
```

```
[04:27:44] Explicit valence for atom # 0 O, 3, is greater than permitted
[04:27:44] ERROR: Could not sanitize molecule ending on line 8
[04:27:44] ERROR: Explicit valence for atom # 0 O, 3, is greater than permitted
[04:27:44]

****
Post-condition Violation
Element 'Qq' not found
Violation occurred on line 90 in file /Users/dalke/ftps/rdkit-Release_2016_09_3/Code/
↪GraphMol/PeriodicTable.h
Failed Expression: anum > -1
****

[04:27:44] Unexpected error hit on line 14
[04:27:44] ERROR: moving to the begining of the next molecule
>>> ids
['nitrogen']
```

As expected, RDKit could only extract one record of the three. It helpfully points out the line number of the records it couldn't parse (lines 8 and 14)

Now I'll do the same using chemfp's `rdkit_toolkit` interface and the default error handler, which is `strict`:

```
>>> from chemfp import rdkit_toolkit
>>> ids = []
>>> for id, mol in rdkit_toolkit.read_ids_and_molecules("bad_data.sdf"):
...   ids.append(id)
...
[15:41:07] Explicit valence for atom # 0 O, 3, is greater than permitted
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "chemfp/_rdkit_toolkit.py", line 1252, in _iter_read_sdf_structures
    error_handler.error("Could not parse molecule block", location)
  File "chemfp/io.py", line 87, in error
    _compat.raise_tb(ParseError(msg, location), None)
  File "<string>", line 1, in raise_tb
chemfp.ParseError: Could not parse molecule block, file 'bad_data.sdf', line 1,␣
↪record #1: first line is 'trivalent'
```

It stops at the first error and raise an exception. The exception contains some information about the error location, including the filename, line number, record number, and the contents of the first line of the file.

How does chemfp get that information? Under the covers chemfp uses its own parser, from the *`text_toolkit`* to read each record, then passes that record to RDKit to turn the record into a molecule. This gives chemfp a bit more control over error reporting. Originally this was also faster than using RDKit's own `ForwardSDMolSupplier`, but now chemfp is about 10% slower. A future implementation may offer a run-time choice of which implementation to use, in case you want better performance at the expense of less detailed error information.

Pass in either "`ignore`" or "`report`" as the `errors` option if you want chemfp to skip records with an error keep on processing. I'll use "`report`" to show what the error reporting looks like:

```
>>> from chemfp import rdkit_toolkit
>>> ids = []
>>> for id, mol in rdkit_toolkit.read_ids_and_molecules(
...     "bad_data.sdf", errors="report"):
...   ids.append(id)
...
[15:50:23] Explicit valence for atom # 0 O, 3, is greater than permitted
```

```
ERROR: Could not parse molecule block, file 'bad_data.sdf', line 1, record #1: first␣
↪line is 'trivalent'. Skipping.
[15:50:23]

****
Post-condition Violation
Element 'Qq' not found
Violation occurred on line 90 in file /Users/dalke/ftps/rdkit-Release_2016_09_3/Code/
↪GraphMol/PeriodicTable.h
Failed Expression: anum > -1
****

ERROR: Could not parse molecule block, file 'bad_data.sdf', line 10, record #2: first␣
↪line is 'Q-record'. Skipping.
>>> ids
[u'nitrogen']
```

RDKit's own error messages from ForwardSDMolSupplier, like "Unexpected error hit on line 14" / "moving to the begining of the next molecule", have disappeared, because chemfp handles record extraction. The sanitization error message about explicit valence remains because RDKit still does that work.

Note also that under Python 2.7 chemfp returns a Unicode string for the id, rather than the byte string that the native RDKit API returns.

That was RDKit. What about Open Babel?

```
>>> from __future__ import print_function
>>> from chemfp import openbabel_toolkit
>>> with openbabel_toolkit.read_ids_and_molecules(
...     "bad_data.sdf", "sdf", errors="strict") as reader:
...   for id, mol in reader:
...     print("Read", repr(id), "first atom:", mol.GetAtom(1).GetAtomicNum())
...
Read u'trivalent' first atom: 8
==============================
*** Open Babel Warning  in GetAtomicNum
  Cannot understand the element label Qq.
Read u'Q-record' first atom: 0
Read u'nitrogen' first atom: 7
```

Open Babel reads all three records even in strict mode, though that odd "Qq" atom causes it to print a warning message. (It is hard for chemfp to capture that warning message, so I don't.) Interestingly, it turns that atom into a "*" atom, with atomic number 0. To double check, I'll read the list of molecules, then write them all out as SMILES:

```
>>> mols = []
>>> with openbabel_toolkit.read_molecules("bad_data.sdf") as reader:
...   mols = [openbabel_toolkit.copy_molecule(mol) for mol in reader]
...
==============================
*** Open Babel Warning  in GetAtomicNum
  Cannot understand the element label Qq.
>>> len(mols)
3
>>> with openbabel_toolkit.open_molecule_writer(None, "smi") as writer:
...   writer.write_molecules(mols)
...
C#[O] trivalent
* Q-record
```

```
[N] nitrogen
```

OEChem also parses that "Qq" record as an atom with atomic number of 0, though it doesn't even give me a warning message:

```
>>> from chemfp import openeye_toolkit
>>> with openeye_toolkit.read_ids_and_molecules(
...     "bad_data.sdf", errors="strict") as reader:
...   for id, mol in reader:
...     print("Read", repr(id), [a.GetAtomicNum() for a in mol.GetAtoms()])
...
Read u'trivalent' [8, 6]
Read u'Q-record' [0]
Read u'nitrogen' [7]
```

I totally didn't expect the toolkits to parse an unknown atom type like "Qq"!

In any case, OEChem will skip records which it could not parse, and there's no easy way for chemfp to get that information, so in practice the "strict" and "report" options are meaningless.

Ignore errors in create_string() and create_bytes()

In this section you'll learn how to ignore errors when converting a molecule into a string or byte record.

Some molecules cannot be represented in some formats. The easiest example is the molecule from the SMILES "*", which contains a single atom with the atomic number 0 and cannot be represented in InChI:

```
>>> from chemfp import rdkit_toolkit as T
>>> mol = T.parse_molecule("*", "smistring")
>>> T.create_string(mol, "smistring")
u'[*]'
>>> T.create_string(mol, "inchistring")
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "chemfp/rdkit_toolkit.py", line 428, in create_string
    return _toolkit.create_string(mol, format, id, writer_args, errors)
        ....
  File "chemfp/io.py", line 87, in error
    _compat.raise_tb(ParseError(msg, location), None)
  File "<string>", line 1, in raise_tb
chemfp.ParseError: RDKit cannot create the InChI string
```

By default the *chemfp.toolkit.create_string()* and *chemfp.toolkit.create_bytes()* functions will raise an exception if the molecule cannot be converted into the given record format. Use the `errors` parameter to specify that behavior. Just like with file reading, the default value is "`strict`", "`ignore`" will return None if there was an error, and "`report`" will return None and also print some information about the failure to stderr. (Error reporting is more useful for writing

The following uses "`ignore`":

```
>>> from __future__ import print_function
>>> import chemfp
>>> for toolkit in ("openbabel", "rdkit", "openeye"):
...   T = chemfp.get_toolkit(toolkit)
...   mol = T.parse_molecule("*", "smistring")
...   result = T.create_string(mol, "inchistring", errors="ignore")
...   print(toolkit, "returned", repr(result))
```

```
...
============================
*** Open Babel Warning  in InChI code
  #0 :Unknown element(s): Xx
============================
*** Open Babel Error  in InChI code
  InChI generation failed
openbabel returned None
[18:10:46] ERROR: Unknown element(s): *
rdkit returned None
Warning: Unable to create InChI from molecule '' with wild card atoms:␣
↪OEAtomBase::GetAtomicNum() == 0.
openeye returned None
```

The following uses "`report`". You can see the only addition is the new line 'ERROR: Open Babel cannot create the InChI string. Skipping.' For a bit of variation, I also changed things to use `create_bytes` instead of `create_string`:

```
>>> from chemfp import openbabel_toolkit as T
>>> mol = T.parse_molecule("*", "smistring")
>>> result = T.create_bytes(mol, "inchistring", errors="report")
============================
*** Open Babel Warning  in InChI code
  #0 :Unknown element(s): Xx
============================
*** Open Babel Error  in InChI code
  InChI generation failed
ERROR: Open Babel cannot create the InChI string. Skipping.
>>> result is None
True
```

Ignore errors when writing molecules

In this section you'll learn how to ignore errors and improve error reporting when writing a file, rather than accept the default of raising an exception and stopping. You will need a copy of ChEBI_lite.sdf.gz.

It's not unusal for there to be a few input records which cannot be parsed into a molecule. It's much less common to come across a molecule which cannot be turned into a record. The SMILES and SD file formats are able to handle a wide range of chemistry. Even R-groups, which can't directly be expressed as SMILES, can be represented in one of several conventions, like `[*:1]` for R1.

There are no such conventions for InChI. As you saw in the previous section, it's easy to make a molecule to InChI converter fail if the structure contains a "*" atom.

The functions `chemp.toolkit.open_molecule_writer()`, `chemp.toolkit.open_molecule_writer_to_string()`, and `chemp.toolkit.open_molecule_writer_to_bytes()` return a molecule writer. This can be used to write a single molecule at a time, or to write molecule multiples from an iterator.

What happens if I try to convert the ChEBI file into an InChI file?

```
>>> reader = T.read_molecules("ChEBI_lite.sdf.gz")
>>> writer = T.open_molecule_writer("chebi.inchi")
>>> writer.write_molecules(reader)
Warning: Unable to create InChI from molecule '' with wild card atoms:␣
↪OEAtomBase::GetAtomicNum() == 0.
Traceback (most recent call last):
```

```
  File "<stdin>", line 1, in <module>
  File "chemfp/base_toolkit.py", line 265, in write_molecules
    _compat.raise_tb(err[0], err[1])
  File "chemfp/_openeye_toolkit.py", line 354, in _gen_write_inchi_structures
    error_handler.error(errmsg, location)
  File "chemfp/io.py", line 87, in error
    _compat.raise_tb(ParseError(msg, location), None)
  File "<string>", line 1, in raise_tb
chemfp.ParseError: OEChem cannot create the InChI string, file 'chebi.inchi', record
↪#3
```

The third record could not be converted to an InChI string, and the warning message that OEChem printed to the termal shows that the molecule contained a wildcard atom, that is, the "*" atom. But, did it really?

```
>>> reader = T.read_molecules("ChEBI_lite.sdf.gz")
>>> for i in range(3):
...   mol = next(reader)
...   print(T.create_string(mol, "smistring"))
...
c1cc(c(cc1[C@@H]2[C@@H](Cc3c(cc(cc3O2)O)O)O)O)O
C[C@]12CC[C@H](C1)C(C2=O)(C)C
*C(=O)OC(CO)CO[R1]
```

That shows "R1", not an R-group. What's going on? "R1" isn't even a valid SMILES.

This is an OEChem extension to SMILES. The default output SMILES flavor includes the flag "RGroups", which

> [c]ontrols whether atoms with atomic number zero (as determined by the OEAtomBase::GetAtomicNum method), and a non-zero map index (as determined by the OEAtomBase::GetMapIdx method) should be displayed using the [R1] notation. In this notation, the integer value following the R corresponds to the atom's map index. When this flag isn't set, such atoms are written in the Daylight convention [*:1]. – OEChem documenation

I'll redo the loop but this time disable the RGroup using the writer_args option to set the flavor to "`Default,-RGroups`", that is, the default value but without RGroups being set:

```
>>> reader = T.read_molecules("ChEBI_lite.sdf.gz")
>>> for i in range(5):
...   mol = next(reader)
...   print(T.create_string(mol, "smistring",
...       writer_args={"flavor": "Default,-RGroups"}))
...
c1cc(c(cc1[C@@H]2[C@@H](Cc3c(cc(cc3O2)O)O)O)O)O
C[C@]12CC[C@H](C1)C(C2=O)(C)C
*C(=O)OC(CO)CO[*:1]
C[C@]12CC[C@@H]3c4ccc(cc4CC[C@H]3[C@@H]1C[C@H](C2=O)O)O
c1cc(c(c(c1)Cl)C#N)Cl
```

That indeed gives `[*:1]` which is the wildcard atom that InChI complains about.

The molecule writers support the same `errors` option as the molecule readers. The default value is "`strict`", which means to raise an exception. To ignore errors, use "`ignore`", and to ignore errors except to report a message to standard out, use "`report`".

```
>>> from chemfp import openeye_toolkit as T # use your toolkit of choice
>>> reader = T.read_ids_and_molecules("ChEBI_lite.sdf.gz", id_tag="ChEBI ID", errors=
↪"ignore")
>>> writer = T.open_molecule_writer("chebi.inchi", errors="report")
>>> writer.write_ids_and_molecules(reader)
```

The first few and last few lines of output are:

```
Warning: Unable to create InChI from molecule '' with wild card atoms:␣
↪OEAtomBase::GetAtomicNum() == 0.
ERROR: OEChem cannot create the InChI string, file 'chebi.inchi', record #3. Skipping.
Warning: Unsupported Sgroup information ignored
Warning: Unsupported Sgroup information ignored
Warning: Unable to create InChI from molecule '' with wild card atoms:␣
↪OEAtomBase::GetAtomicNum() == 0.
ERROR: OEChem cannot create the InChI string, file 'chebi.inchi', record #13.␣
↪Skipping.
Warning: Stereochemistry corrected on atom number 2 of
Warning: Unable to create InChI from molecule '' with wild card atoms:␣
↪OEAtomBase::GetAtomicNum() == 0.
ERROR: OEChem cannot create the InChI string, file 'chebi.inchi', record #133.␣
↪Skipping.
    ...
ERROR: OEChem cannot create the InChI string, file 'chebi.inchi', record #94443.␣
↪Skipping.
Warning: Stereochemistry corrected on atom number 8 of
Warning: Stereochemistry corrected on atom number 13 of
Warning: Stereochemistry corrected on atom number 36 of
```

where the lines starting "ERROR: OEChem" come from chemfp, and the others come from OEChem at a lower-level. (Alas, the "report" isn't as helpful as it should be. I would like it to include the output id in the error message, but all it gives is the record number. Perhaps it will be in the next release?)

All told, there were 94633 of which 88839 could be written out. I got these numbers from the writer's `location` property (see *Location information: filename, record_format, recno and output_recno*, below). Its `recno` is the number of records sent to the writer, and `output_recno` is the number of records actually written:

```
>>> writer.location.recno
94633
>>> writer.location.output_recno
88839
```

Reader and writer format metadata

In this section you'll learn about the format `metadata` attribute of the readers and writers. You will need Compound_014550001_014575000.sdf.gz from PubChem if you want to reproduce this for yourself.

Each reader and writer has a `metadata` attribute, which stores some information about the parameters used to open it:

```
>>> from chemfp import rdkit_toolkit as T
>>> reader = T.read_molecules("Compound_014550001_014575000.sdf.gz")
>>> reader.metadata
FormatMetadata(filename='Compound_014550001_014575000.sdf.gz',
record_format='sdf', args={'strictParsing': True, 'removeHs': True, 'sanitize': True})
>>> writer = T.open_molecule_writer(None, "sdf")
>>> writer.metadata
FormatMetadata(filename='<stdout>', record_format='sdf', args={'kekulize': True,
↪'includeStereo': False})
```

The metadata for a structure reader and writer is a *chemfp.base_toolkit.FormatMetadata* instances, and not the *chemfp.Metadata* for a fingerprint reader and writer.

The *filename* attribute is best effort at a string representation of the source or destination. It can either be the original filename (if there is one), the strings "<stdin>" or "<stdout>" for stdin/stout, the string "<string>" if reading or writing to memory, the source or destination's "name" attribute if a file object, or None if all else fails.

The *record_format* attribute is the format name for the record, which is the same as the input file format except without any compression. As you can see in the above example, the "sdf.gz" reader has a `record_format` of "sdf". This parameter is useful when you want use the *text_toolkit* to extract records because you pass the text reader's record format as the *format* for the chemistry toolkit's *toolkit.parse_molecule()*.

The *args* attribute is the processed reader_args or writer_args, without any namespacing. For now it's mostly available for debugging purposes, so you can see how the toolkit layer actually processed your arguments. In the future there will be a way to turn this into a text settings dictionary.

Location information: filename, record_format, recno and output_recno

In this section you'll learn the basics of the *chemfp.io.Location* API, you'll learn how to get the location object for each reader and writer, and you'll learn about the `recno` and `output_recno` location attributes.

(See the next section for details about the `lineno`, `offsets`, `record`, and other location properties which are not available for every toolkit format.)

The reader and writers track information about the current state of the reader and writer. Some of this information is more generally useful, and available through the `location` attribute of each reader and writer:

```
>>> from chemfp import rdkit_toolkit as T # use your toolkit of choice
>>> content = "C methane\nO=O oxygen\n"
>>> reader = T.read_molecules_from_string(content, "smi")
>>> reader.location
Location('<string>')
>>> loc = reader.location
>>> loc.filename
'<string>'
>>> loc.record_format
'smi'
```

If there is no actual filename then `filename` is "<string>" for string-based I/O, "<stdin>" when reading from stdin, and "<stdout>" when writing to stdout. (The latter two occur when the source or destination parameter, respectively, are None.) The `record_format` is the record format name, without any compression suffix:

```
>>> writer = T.open_molecule_writer("example.sdf.gz")
>>> writer.location.filename
'example.sdf.gz'
>>> writer.location.record_format
'sdf'
>>> writer.close()
```

All of the toolkit readers and writers support the `recno` location property, which is the number of records which have been read or written. A recno of 0 means that no records have been read:

```
>>> reader = T.read_molecules_from_string(content, "smi")
>>> loc = reader.location
>>> loc.recno
0
>>> next(reader)
<rdkit.Chem.rdchem.Mol object at 0x10fb06e50>
>>> loc.recno
1
>>> next(reader)
```

```
<rdkit.Chem.rdchem.Mol object at 0x10fb06ec0>
>>> loc.recno
2
```

While you could use the `recno` property for simple enumeration, as in the folllowing:

```
>>> from __future__ import print_function
>>> with T.read_ids_and_molecules_from_string(content, "smi") as reader:
...     loc = reader.location
...     for id, mol in reader:
...         print("record number:", loc.recno, "id:", id)
...
record number: 1 id: methane
record number: 2 id: oxygen
```

I would prefer that you write it with the "enumerate()" function, as in:

```
>>> with T.read_ids_and_molecules_from_string(content, "smi") as reader:
...   for recno, (id, mol) in enumerate(reader, 1):
...     print("record number:", recno, "id:", id)
...
record number: 1 id: methane
record number: 2 id: oxygen
```

The enumerate() function is both faster and more expected for this sort of code. The `recno` property exists more to help with error reporting, and to report summary information, like:

```
>>> print("Read", reader.location.recno, "records")
Read 2 records
```

The output writers distinguish between `recno`, which is the number of molecules that chemfp tried to save, and `output_recno`, which is the number of molecules that could actually be saved. This occurs because some molecules cannot be written to a given format, like the SMILES "*" which has no InChI representation:

```
>>> from chemfp import openbabel_toolkit
>>> writer = openbabel_toolkit.open_molecule_writer("example.inchi")
>>> parse_molecule = openbabel_toolkit.parse_molecule
>>> writer.write_molecule(parse_molecule("c1ccccc1O", "smistring"))
>>> writer.location.recno
1
>>> writer.location.output_recno
1
>>> writer.write_molecule(parse_molecule("*", "smistring"))
==============================
*** Open Babel Warning  in InChI code
  #0 :Unknown element(s): Xx
==============================
*** Open Babel Error  in InChI code
  InChI generation failed
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "/Users/dalke/cvses/cfp-3x/docs/tmp/chemfp/base_toolkit.py", line 253, in␣
↪write_molecule
    _compat.raise_tb(err[0], err[1])
  File "<string>", line 1, in raise_tb
  File "/Users/dalke/cvses/cfp-3x/docs/tmp/chemfp/_openbabel_toolkit.py", line 1314,␣
↪in _gen_write_delimited_structures
    % (format_name,), location)
```

```
  File "/Users/dalke/cvses/cfp-3x/docs/tmp/chemfp/io.py", line 87, in error
    _compat.raise_tb(ParseError(msg, location), None)
  File "<string>", line 1, in raise_tb
chemfp.ParseError: Open Babel cannot create the InChI string, file 'example.inchi',␣
↪record #2
>>> writer.location.recno
2
>>> writer.location.output_recno
1
```

Location information: record position and content

In this section you'll learn how to get position information for each record and information about the content of each record. You will need the RDKit toolkit or Open Babel toolkit. (Unfortunately for me, OEChem doesn't have a way to get this information, and my hybrid parser with improved error reporting proved to be much slower than OEChem's native performance.) You will also need Compound_014550001_014575000.sdf.gz from PubChem.

(See the previous section for details about the `filename`, `record_format`, `recno` and `output_recno` location properties, which are available for every toolkit format.)

Sometimes you want to know where a record is located in a file. You might want to report that the unusable record started on line 12345 of a given file, or you might want to index a file to implement random access lookup.

The underlying toolkits do not implement this functionality. Instead, chemfp includes its own SMILES and SDF file readers. These know enough about the formats to extract a single record, then pass the record to the toolkit to turn into a molecule. This lets chemfp track the line number of the start of the record, its byte range, the text of the current record, and other details.

Timings show that the hybrid parser for the SMILES formats are no slower than the native RDKit and Open Babel readers, and that the hybrid SDF parser a bit slower than RDKit's native parser (about 10%) and slower than Open Babel's native parser. In all cases, OEChem native parsers leave chemfp in the dust.

As a consequence, the *rdkit_toolkit* and *openbabel_toolkit* SMILES readers track more detailed record information, but the *openeye_toolkit* one does not. (The *text_toolkit* of course always tracks that information.) Here is an example which works for rdkit_toolkit and openbabel_toolkit:

```
>>> from __future__ import print_function
>>> from chemfp import openbabel_toolkit as T # or rdkit_toolkit
>>> content = "C methane\nO=O oxygen\n"
>>> reader = T.read_ids_and_molecules_from_string(content, "smi")
>>> loc = reader.location
>>> for id, mol in reader:
...     print("id:", repr(id), "lineno:", loc.lineno, "byte range:", loc.offsets)
...     print("   record content:", repr(loc.record))
...     print("   first line:", repr(loc.first_line))
...
id: 'methane' lineno: 1 byte range: (0, 10)
   record content: b'C methane\n'
   first line: 'C methane'
id: 'oxygen' lineno: 2 byte range: (10, 21)
   record content: b'O=O oxygen\n'
   first line: 'O=O oxygen'
>>> content[0:10]
'C methane\n'
>>> content[10:21]
'O=O oxygen\n'
```

(Note: if the input record is a Unicode string then it will be converted into a UTF-8 encoded byte string. The start and end positions are coordinates in the encoded byte string, not the text string.)

The *location* instance of the rdkit_toolkit SDF reader gives access to many details about the current parser state:

```
>>> from chemfp import rdkit_toolkit
>>> reader = rdkit_toolkit.read_molecules("Compound_014550001_014575000.sdf.gz")
>>> next(reader)
<rdkit.Chem.rdchem.Mol object at 0x1104c9830>
>>> reader.location.lineno
1
>>> reader.location.offsets
(0, 4227)
>>> reader.location.first_line
'14550001'
>>> next(reader)
<rdkit.Chem.rdchem.Mol object at 0x1104c97c0>
>>> reader.location.lineno
166
>>> reader.location.offsets
(4227, 8399)
>>> reader.location.first_line
'14550002'
```

The openbabel_toolkit and openeye_toolkit implementations by default don't track this level of detail, because their native readers are faster than when I can manage in a hybrid reader. Consequently, those values are None:

```
>>> from chemfp import openbabel_toolkit
>>> reader = openbabel_toolkit.read_molecules("Compound_014550001_014575000.sdf.gz")
>>> next(reader)
<openbabel.OBMol; proxy of <Swig Object of type 'OpenBabel::OBMol *' at 0x1106a9030> >
>>> print(reader.location.lineno)
None
>>> print(reader.location.offsets)
None
>>> print(reader.location.first_line)
None
```

There is experimental support to use Open Babel in hybrid mode. The *reader_args* supports an "implementation" option. The default of None, or "openbabel", tells chemfp to use Open Babel's native parser, while specifying "chemfp" tells it to use chemfp's own SDF record parser:

```
>>> openbabel_toolkit.get_format("sdf").get_default_reader_args()
{'implementation': None, 'perceive_0d_stereo': False, 'perceive_stereo': False,
↪'options': None}
>>> reader = openbabel_toolkit.read_molecules("Compound_014550001_014575000.sdf.gz",
...                reader_args={"implementation": "chemfp"})
>>> next(reader)
<openbabel.OBMol; proxy of <Swig Object of type 'OpenBabel::OBMol *' at 0x1106a9de0> >
>>> reader.location.lineno
1
>>> reader.location.offsets
(0, 4227)
>>> reader.location.first_line
'14550001'
```

If user-defined selection of the back-end implementation works well, I may add similar support for the openeye_toolkit, for those who want the increased level of location detail despite the large performance impact.

The RDKit "sdf" reader always uses the hybrid. This is for historical reasons. The hybrid solution was once always faster than the native ForwardSDMolSupplier. That has since changed, and ForwardSDMolSupplier is about 10% faster. At some point I will add an 'implementation' option so you can switch between performance and improved error reporting.

Writing your own error handler (Experimental)

In this section you'll learn how to write your own error handler. This is an advanced topic. Bear in mind that this is highly experimental and very likely to change. I hope you can provide feedback about how to improve it.

In earlier sections you learned that when the errors parameter is "strict", the parser will raise an exception if there's a problem with a record. When it's "ignore", the record parsers return None as the molecule, while the file and string readers skip the failing record. When it's "report", the result is the same as "ignore" except that extra information about the failure is written to stderr.

The errors parameter can also take an object which implements the "errors()" method as in the following:

```
import sys
class OopsHandler(object):
    def error(self, msg, location=None):
        if location is None:
            sys.stderr.write("Oops! %s. Skipping.\n" % (msg,))
        else:
            sys.stderr.write("Oops! %s, %s. Skipping.\n" % (msg, location.where()))
```

The `msg` is a string describing the error, and `location` contains the *chemfp.io.Location* for the given record. Here's what it looks like in action:

```
>>> from __future__ import print_function
>>> import sys
>>> from chemfp import rdkit_toolkit as T
>>> T.parse_molecule("Q", "smistring", errors=OopsHandler())
>>> T.parse_molecule("Q", "smistring", errors=OopsHandler())
[13:21:58] SMILES Parse Error: syntax error for input: 'Q'
Oops! RDKit cannot parse the SMILES string 'Q'. Skipping.
>>> for mol in T.read_molecules_from_string("Q Q-ane\nC methane\n", "smi",
...                errors=OopsHandler()):
...   print("Processed", mol)
...
[13:22:47] SMILES Parse Error: syntax error for input: 'Q'
Oops! RDKit cannot parse the SMILES 'Q', file '<string>', line 1, record #1: first␣
↪line is 'Q Q-ane'. Skipping.
Processed <rdkit.Chem.rdchem.Mol object at 0x1013f0990>
```

The location's *where()* method tries to give useful information based on the location's filename, line number, record number, and the first line of the record (up to the first 40 characters).

It's easy to see how to modify this to send the errors to a logger, or save them up to display in a GUI.

For the hybrid parsers, which give access to the raw record, you can do more advanced processing, like extract the title lines of any SDF record which RDKit can't handle. The following will make an SDF-formatted string containing three records, where the second record is a 5-valent nitrogren that RDKit can't parse. It will then try to parse the string, and store the ids for records which couldn't be parsed.

```
from __future__ import print_function
from rdkit import Chem
from chemfp import rdkit_toolkit
```

```
# Use RDKit to make an SD file which RDKit cannot parse.
methane = rdkit_toolkit.parse_molecule("C methane", "smi")
# Bypass normal sanitization so RDKit will read 5-valent nitrogens
pentavalent_n = rdkit_toolkit.parse_molecule("CN(C)(C)(C)C pentavalent N",
        "smi", reader_args={"sanitize": False})

Chem.SanitizeMol(pentavalent_n, Chem.SanitizeFlags.SANITIZE_SETHYBRIDIZATION)
oxygen = rdkit_toolkit.parse_molecule("O=O oxygen", "smi")

# Use the three molecules to make an SD file as a string
with rdkit_toolkit.open_molecule_writer_to_string("sdf") as writer:
  writer.write_molecules([methane, pentavalent_n, oxygen])

sdf_content = writer.getvalue()

# User-defined error handler
class CaptureIds(object):
  def __init__(self):
    self.ids = []
  def error(self, msg, location):
    self.ids.append(location.first_line)

capture_ids = CaptureIds()

for mol in rdkit_toolkit.read_molecules_from_string(sdf_content, "sdf",
           errors=capture_ids):
    pass

print("Could not parse:", capture_ids.ids)
```

The content of `sdf_content` is:

```
methane
     RDKit

  1  0  0  0  0  0  0  0  0  0999 V2000
    0.0000    0.0000    0.0000 C   0  0  0  0  0  0  0  0  0  0  0  0
M  END
$$$$
pentavalent N
     RDKit

  6  5  0  0  0  0  0  0  0  0999 V2000
    0.0000    0.0000    0.0000 C   0  0  0  0  0  0  0  0  0  0  0  0
    0.0000    0.0000    0.0000 N   0  0  0  0  0  0  0  0  0  0  0  0
    0.0000    0.0000    0.0000 C   0  0  0  0  0  0  0  0  0  0  0  0
    0.0000    0.0000    0.0000 C   0  0  0  0  0  0  0  0  0  0  0  0
    0.0000    0.0000    0.0000 C   0  0  0  0  0  0  0  0  0  0  0  0
    0.0000    0.0000    0.0000 C   0  0  0  0  0  0  0  0  0  0  0  0
  1  2  1  0
  2  3  1  0
  2  4  1  0
  2  5  1  0
  2  6  1  0
M  END
$$$$
oxygen
     RDKit
```

```
  2  1  0  0  0  0  0  0  0  0999 V2000
    0.0000    0.0000    0.0000 O   0  0  0  0  0  0  0  0  0  0  0  0
    0.0000    0.0000    0.0000 O   0  0  0  0  0  0  0  0  0  0  0  0
  1  2  2  0
M  END
$$$$
```

and the output from the above is:

```
[13:26:45] Explicit valence for atom # 1 N, 5, is greater than permitted
Could not parse: [u'pentavalent N']
```

The *fingerprint type documentation* includes another example of how to write an error handler.

A Babel-like structure format converter

In this section you'll learn how to use the chemfp toolkit API to create a Babel-like structure file format converter. This section goes into more details of how to develop real-world software using chemfp.

Pat Walters and Matt Stahl started Babel in the 1990s as a command-line program to convert from one chemical structure format to another. This developed over the years, and after a major rewrite became the LGPL toolkit "OELib", OpenEye's first commercial chemistry toolkit. OpenEye's next rewrite lead to OEChem, a proprietary chemistry toolkit. OELib was still available, and others continued to develop it. It became Open Babel, and structure file format conversion is still Open Babel's forte.

A full Babel-like program includes features to add and remove hydrogens of different sorts, select or reject structures based on substructure or other features, add 2D or 3D coordinates, and more. You cannot use chemfp for that. All chemfp can do is read structure files into a given toolkit's molecule object, and write molecule objects to a given format.

Even that basic ability is useful. I'll explain how to write such a converter yourself. I'll use as my example file the following, "example.smi":

```
c1ccccc1O phenol
C methane
O=O molecular oxygen
```

Here's a minimal conversion program to convert the above into "example.sdf":

```
from chemfp import rdkit_toolkit as T  # use your toolkit of choice

reader = T.read_molecules("example.smi")
writer = T.open_molecule_writer("example.sdf")
writer.write_molecules(reader)
```

That code depends on Python's garbage collection to close the output file handle. This is fine for a script, but a longer running program may want to have more explicit control over closing the file handle and use a context manager (see *Reader and writer context managers*):

```
from chemfp import rdkit_toolkit as T # use your toolkit of choice

with T.read_molecules("example.smi") as reader:
    with T.open_molecule_writer("example.sdf") as writer:
        writer.write_molecules(reader)
```

With that we have enough to build our first Babel program, which takes the input and output filenames on the command-line. I'll call this program "cbabel.py", for "chemfp babel", and have it implement the command-line

```
usage: cbabel.py [-h] input_filename output_filename
```

I'll use argparse from Python's standard library to handle command-line argument processing. The "`nargs=1`" in the following says that the *input_filename* and *output_filename* must exist, and only one filename is allowed. Argparse will save those in a list of size 1, which is why I use [0] to get the actual string I'm interested in:

```
import argparse
from chemfp import rdkit_toolkit as T

parser = argparse.ArgumentParser(
    description = "A minimial chemical structure file converter"
    )
parser.add_argument("input_filename", nargs=1, help="input filename")
parser.add_argument("output_filename", nargs=1, help="output filename")

args = parser.parse_args()

with T.read_molecules(args.input_filename[0]) as reader:
    with T.open_molecule_writer(args.output_filename[0]) as writer:
        writer.write_molecules(reader)
```

I'll convert the SMILES into canonical SMILES:

```
% python cbabel.py example.smi example.can
% cat example.can
Oc1ccccc1 phenol
C methane
O=O molecular oxygen
```

The only change is that the phenol went from `c1ccccc1O` to `Oc1ccccc1`.

I'll add the ability to read from stdin and stdout. I'll say that if the input filename is "-" then it will read from stdin, and if the output filename is "-" then it will write to stdout. (If you have a file named "-" then you'll have to specify "./-" to read or write to it.):

```
import argparse
from chemfp import rdkit_toolkit as T

parser = argparse.ArgumentParser(
    description = "A minimial chemical structure file converter"
    )
parser.add_argument("input_filename", nargs=1, help="input filename")
parser.add_argument("output_filename", nargs=1, help="output filename")

args = parser.parse_args()

# Support "-" as stdin/stdout by mapping it to None,
# which tells chemfp to use stdin/stout
input_filename = args.input_filename[0]
if input_filename == "-":
    input_filename = None

output_filename = args.output_filename[0]
if output_filename == "-":
    output_filename = None
```

```
with T.read_molecules(input_filename) as reader:
    with T.open_molecule_writer(output_filename) as writer:
        writer.write_molecules(reader)
```

There's a problem with this! When the input or output format is None, chemfp can't figure out the format based on the filename, so will assume that it's a SMILES file. When I run the above I get SMILES output:

```
% python cbabel.py example.smi -
Oc1ccccc1 phenol
C methane
O=O molecular oxygen
```

But what if I want SDF output? I need a way to specify the input and output file formats on the command-line. I'll use the `-i` and `-o` options to specify those:

```
from __future__ import print_function
import argparse
from chemfp import rdkit_toolkit as T

parser = argparse.ArgumentParser(
    description = "A minimial chemical structure file converter"
    )
parser.add_argument("-i", metavar="FORMAT", dest="input_format",
                    help="input format name", default=None)
parser.add_argument("-o", metavar="FORMAT", dest="output_format",
                    help="output format name", default=None)
parser.add_argument("input_filename", nargs=1, help="input filename")
parser.add_argument("output_filename", nargs=1, help="output filename")

args = parser.parse_args()

# Support "-" as stdin/stdout by mapping it to None,
# which tells chemfp to use stdin/stout
input_filename = args.input_filename[0]
if input_filename == "-":
    input_filename = None

output_filename = args.output_filename[0]
if output_filename == "-":
    output_filename = None

with T.read_molecules(input_filename, args.input_format) as reader:
    with T.open_molecule_writer(output_filename, args.output_format) as writer:
        writer.write_molecules(reader)
```

Now I can specify that I want stdout to be in SDF format:

```
% python x.py -o sdf example.smi - | head -5
phenol
     RDKit

  7  7  0  0  0  0  0  0  0  0999 V2000
    0.0000    0.0000    0.0000 C   0  0  0  0  0  0  0  0  0  0  0  0
```

In practice, required command-line arguments make life more difficult. For a simple program like this, required arguments are not a problem, but what if I want to add a command to list the available formats? That option doesn't need an input or output filename, but argparse will enforce that requirement anyway.

There are a couple of ways to solve the problem, but the easiest is to let "-" be the default input and output filename. That's easily done by changing:

```
parser.add_argument("input_filename", nargs=1, help="input filename")
parser.add_argument("output_filename", nargs=1, help="output filename")
```

to:

```
parser.add_argument("input_filename", nargs="?", default="-",
                    help="input filename")
parser.add_argument("output_filename", nargs="?", default="-",
                    help="output filename")
```

As a result I can add a new `--list-formats` argument:

```
parser.add_argument("--list-formats", action="store_true",
                    help="list the available file formats")
```

along with the handler for it, which prints information about the toolkit (its name and version string) and each of the formats. Some of the formats, like "smistring", don't have an I/O format (for that, use "smi"), so I need to filter those out. Also, some of the formats, like "inchikey", are output only, and some of the toolkit have formats that they read but don't write, so I give more details about those:

```
args = parser.parse_args()

if args.list_formats:
  print("Available I/O formats for toolkit %s (%s)" % (T.name, T.software))
  for format in T.get_formats():
    if not format.supports_io: # skip formats like "smistring" and "inchistring"
      continue
    if not format.is_output_format:
      msg = " (input only)"
    elif not format.is_input_format:
      msg = " (output only)"
    else:
      msg = ""
    print(" %s%s" % (format.name, msg))
  raise SystemExit(0)
```

For my version of RDKit I get:

```
% python cbabel.py --list-formats
Available I/O formats for toolkit rdkit (RDKit/2016.09.3)
 inchikey (output only)
 usm
 sdf
 can
 smi
 inchi
```

If I used openeye_toolkit instead of rdkit_toolkit I get:

```
Available I/O formats for toolkit openeye (OEChem/20170208)
 mol2h
 xyz
 mopac (output only)
 cdx
 usm
```

```
 smi
 inchikey (output only)
 skc (input only)
 mol2
 sdf
 mmod
 inchi (output only)
 oeb
 mf (output only)
 sln (output only)
 can
 pdb
```

The code so far requires RDKit, but chemfp supports OEChem and Open Babel. Why not add the command-line argument `--toolkit` to specify an alternate toolkit?

I 'll tell argparse that there's a new `--toolkit` argument, which defaults to "rdkit" and also allows "openeye" and "openbabel":

```
parser.add_argument("--toolkit", metavar="NAME", choices=("rdkit", "openeye",
↪"openbabel"),
                    help="toolkit name", default="rdkit")
```

I can no longer import the toolkit directly, which I did as:

```
from chemfp import rdkit_toolkit as T
```

because that line requires that RDKit be installed. Otherwise it will raise an ImportError exception. While that might be reasonable if the user wanted to use the rdkit toolkit, it's not reasonable if the user wanted to use the Open Babel toolkit and didn't care to know that RDKit isn't available.

Instead of a direct import, I'll use *chemfp.get_toolkit()* to get the named toolkit. It raises a ValueError with a useful error message if the toolkit isn't available or is unknown. If that happens, I'll exit, and use that message as the explanation:

```
import chemfp
     # ... skipped many lines
try:
    T = chemfp.get_toolkit(args.toolkit)
except ValueError as err:
    raise parser.error(str(err))
```

After a bit of experimentation I found a small SMILES string which gives a different canonicalization for each of the supported toolkits, which I present as evidence that it really is using a different toolkit:

```
% echo "NCC(N)O example" | python cbabel.py --toolkit openbabel
NCC(O)N example
% echo "NCC(N)O example" | python cbabel.py --toolkit rdkit
NCC(N)O example
% echo "NCC(N)O example" | python cbabel.py --toolkit openeye
C(C(N)O)N example
```

Here's the final code, so you can see how everything works in context:

```
import argparse
import chemfp

parser = argparse.ArgumentParser(
```

```
    description = "A minimial chemical structure file converter"
    )
parser.add_argument("--toolkit", metavar="NAME", choices=("rdkit", "openeye",
↪"openbabel"),
                    help="toolkit name", default="rdkit")
parser.add_argument("-i", metavar="FORMAT", dest="input_format",
                    help="input format name", default=None)
parser.add_argument("-o", metavar="FORMAT", dest="output_format",
                    help="output format name", default=None)
parser.add_argument("input_filename", nargs="?", default="-",
                    help="input filename")
parser.add_argument("output_filename", nargs="?", default="-",
                    help="output filename")

parser.add_argument("--list-formats", action="store_true",
                    help="list the available file formats")

args = parser.parse_args()

try:
  T = chemfp.get_toolkit(args.toolkit)
except ValueError as err:
  raise parser.error(str(err))


if args.list_formats:
  print("Available I/O formats for toolkit %s (%s)" % (T.name, T.software))
  for format in T.get_formats():
    if not format.supports_io: # skip formats like "smistring" and "inchistring"
      continue
    if not format.is_output_format:
      msg = " (input only)"
    elif not format.is_input_format:
      msg = " (output only)"
    else:
      msg = ""
    print(" %s%s" % (format.name, msg))
  raise SystemExit(0)

# Support "-" as stdin/stdout by mapping it to None,
# which tells chemfp to use stdin/stout
input_filename = args.input_filename[0]
if input_filename == "-":
  input_filename = None

output_filename = args.output_filename[0]
if output_filename == "-":
  output_filename = None

with T.read_molecules(input_filename, args.input_format) as reader:
  with T.open_molecule_writer(output_filename, args.output_format) as writer:
      writer.write_molecules(reader)
```

Amazing how the original four lines of code expands to 55. It would be even more if I added full error reporting instead of letting Python throw an exception on errors.

Speaking of errors, you may want to use hard-coded values of `errors="ignore"` or `errors="report"` to have the parser skip records that the toolkit doesn't understand, or perhaps pass in that information as a command-line

argument named `--errors`, with the possible choices of "strict", "report", or "ignore".

You might also add the `-R` and `-W` options to set the reader args and writer args for the formats, but that's more complicated than I wanted to describe in this context. See the next section for a description of how to do it.

argparse text settings to reader and writer args

In this section you'll learn how to use argparse to handle reader args and writer args in the same style that chemfp does.

The previous section showed how to create a Babel-like structure format conversion program and how to use Python's argparse library for command-line processing. That section was getting too long to describe how to support command-line configuration of the reader args and writer args. In this section I'll start with a smaller version of the same code. This one requires an input filename and an output filename on the command-line, and lets the user specify the toolkit:

```
# I put this into a file named "convert.py"
import argparse
import chemfp

parser = argparse.ArgumentParser(
    description = "Experiment with -R and -W options"
    )
parser.add_argument("--toolkit", metavar="NAME", choices=("rdkit", "openeye",
↪"openbabel"),
                    help="toolkit name", default="rdkit")
parser.add_argument("input_filename", nargs=1, help="input filename")
parser.add_argument("output_filename", nargs=1, help="output filename")

args = parser.parse_args()

T = chemfp.get_toolkit(args.toolkit)
source = args.input_filename[0]
destination = args.output_filename[0]

with T.read_molecules(source) as reader:
  with T.open_molecule_writer(destination) as writer:
    writer.write_molecules(reader)
```

I'll walk through the process of how to add support for the `-R` and `-W` options, to make it possible to say:

```
python convert.py example.smi example.can --toolkit rdkit -R delimiter=space -W␣
↪allBondsExplicit=true
```

How to get from the command-line to reader and writer arguments

This requires a few conversions. I need to turn the command-line arguments into reader and writer text settings dictionaries, then convert the text settings into reader_args and and writer_args dictionaries, before finally passing the reader_args and writer_args to the molecule readers and writers. (See *Convert text settings into reader and writer arguments* for more details about converting text settings to reader and writer arguments.)

I'll use argparse to place the `-R` and `-W` option values into separate lists of `KEY=VALUE` strings, and create a new function which splits them apart on the "=" to get a dictionary of text settings. Then I'll use the *`Format`* object to convert the text settings into the correct *reader_args* and *writer_args*. The steps will look something like this:

```
>>> from chemfp import rdkit_toolkit as T
>>>
```

```
>>> format = T.get_format("smi")  # Specify the format and user-defined settings
>>> reader_settings = ["delimiter=space"]
>>> writer_settings = ["allBondsExplicit=true"]
>>>
>>> # Using a function yet to be defined, convert the list of
... # reader_settings into a dictionary of string values
...
>>> reader_text_settings = parse_text_settings("-R", reader_settings)
>>> reader_text_settings
{'delimiter': 'space'}
>>>
>>> # Ask the format to turn the string values into string objects
...
>>> format.get_reader_args_from_text_settings(reader_text_settings)
{'delimiter': 'space'}
>>>
>>> # Do the same for the writer arguments
...
>>> writer_text_settings = parse_text_settings("-W", writer_settings)
>>> writer_text_settings
{'allBondsExplicit': 'true'}
>>> format.get_writer_args_from_text_settings(writer_text_settings)
{'allBondsExplicit': True}
```

For the actual code the input format may be different than the output format. By the way, if you look closely you'll see how "allBondsExplicit" in the text settings has a value of "true", and the string was converted to the Python object True to be a writer_arg.

To start! First, I need a way to read the list of `-R` and `-W` options. I'll ask argparse to save them into a list, for later post-processing to get the right values:

```
parser.add_argument("-R", metavar="KEY=VALUE", dest="reader_settings", action="append
↪",
                    help="specify a reader argument", default=[])
parser.add_argument("-W", metavar="KEY=VALUE", dest="writer_settings", action="append
↪",
                    help="specify a writer argument", default=[])
```

This will parse all of the `-R` terms, like "`-R delimiter=space`", into the *reader_settings* list, and "`-W allBondsExplicit=true`" into the *writer_settings* list, such that:

```
args.reader_settings == ["delimiter=space"]
args.writer_settings == ["allBondsExplicit=true"]
```

For that matter, it will also support "`-R abc`", and put it into the *reader_settings* list even though it doesn't have a "=" in it. I also need to go through and figure out if any terms are incorrect, and report the problem. I'll make a function for this, along with a parameter so any error message can report if a problem comes from the `-R` or `-W` command-line flag:

```
def parse_text_settings(flag, terms):
  text_settings = {}
  for term in terms:
    left, mid, right = term.partition("=")
    if mid != "=":
      parser.error("%s setting %r must be of the form KEY=VALUE" %
                   (flag, term))
    text_settings[left] = right
  return text_settings
```

```
reader_text_settings = parse_text_settings("-R", args.reader_settings)
writer_text_settings = parse_text_settings("-W", args.writer_settings)
```

This gives me two text settings dictionaries, where the keys and values are both strings. I'll use the respective *Format* object to convert a text setting dictionary into the correct reader and writer arguments dictionary:

```
input_format = T.get_input_format_from_source(source)
reader_args = input_format.get_reader_args_from_text_settings(reader_text_settings)
output_format = T.get_output_format_from_destination(destination)
writer_args = input_format.get_writer_args_from_text_settings(writer_text_settings)
```

All that's left is to pass the *reader_args* and *writer_args* to the reader and writer. Since I already have the input and output format objects, I'll pass those in as well, rather than have them guess again based on the source and destination names:

```
with T.read_molecules(source, input_format, reader_args=reader_args) as reader:
    with T.open_molecule_writer(destination, output_format, writer_args=writer_args)␣
↪as writer:
        writer.write_molecules(reader)
```

Converter with -R and -W support

Here's how it looks when I put it all together:

```
# I put this into a file named "convert.py"
import argparse
import chemfp

parser = argparse.ArgumentParser(
    description = "Experiment with -R and -W options"
    )
parser.add_argument("--toolkit", metavar="NAME", choices=("rdkit", "openeye",
↪"openbabel"),
                    help="toolkit name", default="rdkit")
parser.add_argument("-R", metavar="KEY=VALUE", dest="reader_settings", action="append
↪",
                    help="specify a reader argument", default=[])
parser.add_argument("-W", metavar="KEY=VALUE", dest="writer_settings", action="append
↪",
                    help="specify a writer argument", default=[])
parser.add_argument("input_filename", nargs=1, help="input filename")
parser.add_argument("output_filename", nargs=1, help="output filename")

def parse_text_settings(flag, terms):
  text_settings = {}
  for term in terms:
    left, mid, right = term.partition("=")
    if mid != "=":
      parser.error("%s setting %r must be of the form KEY=VALUE" %
                   (flag, term))
    text_settings[left] = right
  return text_settings


args = parser.parse_args()
```

```
T = chemfp.get_toolkit(args.toolkit)
source = args.input_filename[0]
destination = args.output_filename[0]

input_format = T.get_input_format_from_source(source)
reader_text_settings = parse_text_settings("-R", args.reader_settings)
reader_args = input_format.get_reader_args_from_text_settings(reader_text_settings)

output_format = T.get_output_format_from_destination(destination)
writer_text_settings = parse_text_settings("-W", args.writer_settings)
writer_args = input_format.get_writer_args_from_text_settings(writer_text_settings)


with T.read_molecules(source, input_format, reader_args=reader_args) as reader:
  with T.open_molecule_writer(destination, output_format, writer_args=writer_args) as␣
↪writer:
    writer.write_molecules(reader)
```

Let's see it in action. I'll ask RDKit to include all of the bonds in the output SMILES, including the aromatic bonds, and I'll ask it to use the space character as the SMILES delimiter:

```
% python convert.py example.smi example_output.smi --toolkit rdkit -R delimiter=space␣
↪-W allBondsExplicit=true
% cat example_output.smi
O-c1:c:c:c:c:c:1 phenol
C methane
O=O molecular
```

The lack of "oxygen" in "molecular oxygen" shows that the input SMILES reader used the "space" delimiter instead of the default "to-eol" delimiter, just as I requested.

The `-R` and `-W` settings can also be qualified. (See *Qualified reader and writer parameters names*.) I'll have Open Babel and OEChem use different delimiter styles to get different results:

```
% python convert.py example.smi example_ob_output.smi --toolkit openbabel \
    -R "openbabel.*.delimiter=to-eol"  -R "openeye.*.delimiter=whitespace"
% cat example_ob_output.smi
Oc1ccccc1 phenol
C methane
O=O molecular oxygen
%
% python convert.py example.smi example_oe_output.smi --toolkit openeye \
  -R "openbabel.*.delimiter=to-eol" -R "openeye.*.delimiter=whitespace"
% cat example_oe_output.smi
c1ccc(cc1)O phenol
C methane
O=O molecular
```

List the reader and writer arguments for the given formats

Finally, it's difficult to remember all of the available settings for each input and output format. I'll add a `--list-args` command-line option which shows the available options, and for each option show the current setting, along with an indicator if the current setting is the default value for that format or if the setting comes from the command-line option.

I need argparse to know about the new option:

```
parser.add_argument("--list-args", action="store_true",
                    help="list the available input and output options")
```

and for the rest I replace the last three lines of the earlier code with:

```
if args.list_args:
  # Make a helper function to display the arguments
  def report_args(format, msg, default_args, specified_args):
    print("%s %s:" % (format.name, msg))
    # Merge the arguments; command-line overrides defaults;
    all_args = default_args.copy()
    all_args.update(specified_args)
    for name, value in sorted(all_args.items()):
        # Was the name specified via -R/-W or is it a default?
        where = "from command-line" if name in specified_args else "default value"
        print("      %s: %r  (%s)" % (name, value, where))
  report_args(input_format, "reader arguments (-R)",
              input_format.get_default_reader_args(), reader_args)
  report_args(output_format, "writer arguments (-W)",
              output_format.get_default_writer_args(), writer_args)

else:
  with T.read_molecules(source, input_format, reader_args=reader_args) as reader:
    with T.open_molecule_writer(destination, output_format, writer_args=writer_args)␣
↪as writer:
      writer.write_molecules(reader)
```

(See *Get the default reader_args or writer_args for a format* for more details on the default reader and writer arguments.)

With those changes, the output using the new `--list-args` is:

```
% python convert.py example.smi example_output.smi --toolkit rdkit \
?    -R delimiter=space -W allBondsExplicit=true --list-args
smi reader arguments (-R):
      delimiter: 'space'  (from command-line)
      has_header: False  (default value)
      sanitize: True  (default value)
smi writer arguments (-W):
      allBondsExplicit: True  (from command-line)
      canonical: True  (default value)
      delimiter: None  (default value)
      isomericSmiles: True  (default value)
      kekuleSmiles: False  (default value)
```

Creating a specialized record parser

In this section you'll learn how to make a specialized function to parse an record into a toolkit molecule. This function is somewhat faster than calling the more general purpose *`toolkit.parse_id_and_molecule()`* and might be used when you need to convert a lot of individual records into a molecule.

Sometimes you need to parse a lot of records which don't come from a file. For example, substructure search is typically split into a screening stage based on substructure fingerprints, followed by the atom-by-atom substructure search. The screening stage returns identifiers and the substructure search takes molecules, so in between them is code to look up a record based on its id and convert the result to a molecule.

Assuming a database record API where "`db[id]`" returns the record for a given id, that lookup function might look like this:

```
def get_molecules(db, id_iter, toolkit, format, reader_args=None):
    for id in id_iter:
        record = db[id]
        mol = toolkit.parse_molecule(record, format, reader_args=reader_args)
        yield mol
```

(A more complex implementation should handle when the record id doesn't exist, or can't be converted into a molecule.)

This isn't as fast as it could be. The *toolkit.parse_molecule()* function validates that the *format* and *reader_args* are correct and figures out the right parameters for the underlying toolkit code. It's a waste of time to redo those checks for every single call.

The function also promises that the caller will get a new molecule each time. That promise isn't needed for substructure screening. Timing tests with OEChem show that reusing the same molecule is faster than creating a new one. For example, this OEChem code:

```
mol = OEGraphMol()
for i in xrange(100000):
    OEParseSmiles(mol, "c1ccccc1Oc1ccccc1")
    mol.Clear()
```

takes about 60% of the time as this code:

```
for i in xrange(100000):
    mol = OEGraphMol()
    OEParseSmiles(mol, "c1ccccc1Oc1ccccc1")
```

(Bear in mind that this code isn't doing aromaticity perception, which roughly halves the performance.)

The function *toolkit.make_id_and_molecule_parser()* returns a specialized function to parse records, based on the specified parameters:

```
>>> from chemfp import rdkit_toolkit as T
>>> parser = T.make_id_and_molecule_parser("smi")
>>> parser("c1ccccc1O phenol")
(u'phenol', <rdkit.Chem.rdchem.Mol object at 0x107559980>)
```

For RDKit it's about 10-15% faster to use the specialized function instead of the general purpose *toolkit.parse_molecule()*:

```
>>> from __future__ import print_function
>>> from chemfp import rdkit_toolkit as T
>>> import time
>>>
>>> smiles = "c1ccccc1Oc1ccccc1"
>>> if 1:
...     t1 = time.time()
...     for i in xrange(10000):
...         mol = T.parse_molecule(smiles, "smi")
...     print("Standard time:", time.time()-t1)
...
Standard time: 2.32303786278
>>> parser = T.make_id_and_molecule_parser("smi")
>>> if 1:
...     t1 = time.time()
```

```
...     for i in xrange(10000):
...         id, mol = parser(smiles)
...     print("Specialized time:", time.time()-t1)
...
Specialized time: 2.74086713791
```

The *toolkit.make_id_and_molecule_parser()* function parameters are almost identical to the ones in *toolkit.parse_id_and_molecule()*, and with the same meaning. The only difference is that `make_id_and_molecule_parser` does not support the *record* parameter. Instead, it returns a function which takes the *record* and returns the (id, toolkit molecule) pair:

```
>>> from chemfp import rdkit_toolkit
>>> parser = rdkit_toolkit.make_id_and_molecule_parser(
...     "smi", reader_args={"delimiter": "whitespace"}, errors="ignore")
>>> parser("c1ccccc1O methane 16.04246")
(u'methane', <rdkit.Chem.rdchem.Mol object at 0x108dc9d00>)
>>> parser("Q q-ane")
[14:23:02] SMILES Parse Error: syntax error for input: 'Q'
(None, None)
```

WARNING: The function that `make_id_and_molecule_parser()` returns may reuse the underlying molecule object. Calling the function again may change the molecule returned in previous call:

```
>>> from chemfp import openeye_toolkit
>>> parser = openeye_toolkit.make_id_and_molecule_parser("smi")
>>> id, mol = parser("C")
>>> mol.NumAtoms()
1
>>>
>>> parser("CCC")
(None, <oechem.OEGraphMol; proxy of <Swig Object of type 'OEGraphMolWrapper *' at␣
↪0x10a487600> >)
>>> mol.NumAtoms()
3
```

RDKit doesn't support molecule reuse so rdkit_toolkit returns a new molecule. Open Babel does support reuse and openbabel_toolkit will reuse the molecule. However, my tests using Open Babel show a barely detectable performance improvement if I reuse a molecule vs. creating a new one each time. Future versions of chemfp may change the default, and may add an implementation option to specify if a new molecule should be returned each time.

In multithreaded code you should create a new parser for each thread.

You might have noticed there is no "`make_molecule_parser()`". While it would be useful, it takes time to develop, test, and document, and it wasn't useful enough for this release. Let me know if you would like it in the future.

Molecule API: Get and set the molecule id

In this section you'll learn how to get and set the molecule id for a toolkit molecule.

Sometimes you want to get or set toolkit molecule id. This should be pretty rare because the input routines all support a way to get the identifier in parallel with the molecule, and the output routines all support a way to specify an identifier.

One exception is if you read molecules from an SD file where you want to use one of the SD tag values as the identifier rather than the title line at the top of the record. This can occur with the ChEBI data set:

```
>>> from chemfp import rdkit_toolkit as T
>>> reader = T.read_ids_and_molecules("ChEBI_lite.sdf.gz", id_tag="ChEBI ID")
>>> next(reader)
(u'CHEBI:90', <rdkit.Chem.rdchem.Mol object at 0x10d4a9360>)
>>> id, mol = _
>>> id
u'CHEBI:90'
>>> mol
<rdkit.Chem.rdchem.Mol object at 0x10d4a9360>
>>> mol.GetProp("_Name")
''
>>> print(reader.location.record[:200])

  Marvin  01211310252D

 22 24  0  0  0  0            999 V2000
   -2.8644   -0.2905    0.0000 C   0  0  0  0  0  0  0  0  0  0  0  0
   -2.8656   -1.1176    0.0000 C   0  0  0  0  0  0  0
>>>
```

Note: in Python 3, the location.record is a byte string and the last output line is:

```
>>> print(reader.location.record[:200])
b'\n  Marvin  01211310252D          \n\n 22 24  0  0  0  0            999 V2000
\n   -2.8644   -0.2905    0.0000 C   0  0  0  0  0  0  0  0  0  0  0  0\n   -2.
8656   -1.1176    0.0000 C   0  0  0  0  0  0  0 '
```

The above used the RDKit-specific way to get the special "_Name" property and show that it's the empty string. The location object for the rdkit_toolkit SDF reader is able to show the raw text for the current record. In the above I used the location.record to show that the record indeed has no title line.

I might want to tie that id directly to the molecule. For example, a lot of people write code which assume that the molecule's name or title is the identifier, because only ChEBI and a scant handful of other databases use an alternative convention. You can use chemfp to get the appropriate id, then set the correct molecular property.

If I know it's an RDKit molecule then I could do:

```
mol.SetProp("_Name", id)
```

This is not portable. OEChem and Open Babel call this a "title", and use the molecule's "GetTitle()" and "SetTitle()" accession methods to get and set it. For those toolkits I would need to do:

```
mol.SetTitle(id)
```

As part of chemfp's limited molecule API, each chemfp toolkit layer implements a portable helper function named "get_id()" to get the toolkit-appropriate "identifier", and "set_id()" to set it. The following shows an example of converting the title of a SMILES record to upper-case, and generating the corresponding canonical SMILES:

```
>>> from __future__ import print_function
>>> import chemfp
>>> for toolkit_name in ("rdkit", "openeye", "openbabel"):
...   T = chemfp.get_toolkit(toolkit_name)
...   mol = T.parse_molecule("c1ccccc1O phenol", "smi")
...   T.set_id(mol, T.get_id(mol).upper())
...   smiles = T.create_string(mol, "smi")
...   print(toolkit_name, "->", repr(smiles))
...
rdkit -> u'Oc1ccccc1 PHENOL\n'
```

```
openeye -> u'c1ccc(cc1)O PHENOL\n'
openbabel -> u'Oc1ccccc1 PHENOL\n'
```

Please note that this could be written more succinctly by passing the `id` directly to the *chemfp.toolkit.create_string()* function, as:

```
>>> from __future__ import print_function
>>> import chemfp
>>> for toolkit_name in ("rdkit", "openeye", "openbabel"):
...   T = chemfp.get_toolkit(toolkit_name)
...   id, mol = T.parse_id_and_molecule("c1ccccc1O phenol", "smi")
...   smiles = T. create_string(mol, "smi", id=id.upper())
...   print(toolkit_name, "->", repr(smiles))
...
rdkit -> 'Oc1ccccc1 PHENOL\n'
openeye -> 'c1ccc(cc1)O PHENOL\n'
openbabel -> 'Oc1ccccc1 PHENOL\n'
```

Note: I may add support for an optional id_tag, as in:

```
T.get_id(mol, id_tag="ChEBI id")     # Currently not valid chemfp code!
```

If you think this would be useful, please let me know about your use case.

Finally, if you want the output record as a UTF-8 encoded byte string rather than a Unicode string then use *chemfp.toolkit.create_bytes()* instead of `create_string()`.

Molecule API: Copy a molecule

In this section you'll learn how to make a copy of a native toolkit molecule.

The chemfp file readers may clear and reuse the underlying toolkit molecule. This is a problem if you want to load all of the molecules from a data set into memory:

```
>>> from chemfp import openeye_toolkit
>>> content = "C methane\nO=O oxygen\n"
>>> mols = list(openeye_toolkit.read_molecules_from_string(content, "smi"))
>>> mols
[<oechem.OEGraphMol; proxy of <Swig Object of type 'OEGraphMolWrapper *' at␣
↪0x109776d20> >,
<oechem.OEGraphMol; proxy of <Swig Object of type 'OEGraphMolWrapper *' at␣
↪0x109776d20> >]
>>> mols[0] is mols[1]
True
>>> openeye_toolkit.create_string(mols[0], "smistring")
''
>>> openeye_toolkit.create_string(mols[1], "smistring")
''
```

You can see that the openeye_toolkit reader reuses the same OEGraphMol, and that the molecule is cleared at the end of parsing.

In the future there may be a *reader_args* parameter to tell the reader to make a new molecule for each term. Until that possible future happens, one work-around is to make a copy of the molecule using the respective chemfp toolkit's *toolkit.copy_molecule()* function:

```
>>> from chemfp import openeye_toolkit as T
>>> mols = [T.copy_molecule(mol) for mol in openeye_toolkit.read_molecules_from_
↪string(content, "smi")]
>>> mols
[<oechem.OEGraphMol; proxy of <Swig Object of type 'OEGraphMolWrapper *' at␣
↪0x10b31e930> >,
<oechem.OEGraphMol; proxy of <Swig Object of type 'OEGraphMolWrapper *' at␣
↪0x10b31e6f0> >]
>>> mols[0] is mols[1]
False
>>> T.create_string(mols[0], "smistring")
'C'
>>> T.create_string(mols[1], "smistring")
'O=O'
```

The various writers may also modify the molecule, for example, by temporarily changing the molecule id or by reperceiving aromaticity. If this is a problem then you can use the copy_molecule() as a way to work around it.

This is definitely a work-around solution because it's currently impossible to know if a copy is needed or not. The fail-safe solution is to always copy, which will lead to extra copies and slower code when using the rdkit_toolkit. Other more complicated workarounds might be faster, but the real solution that I hope to implement in the future is to specify the requested behavior as a parameter.

Molecule API: Working with SD tags

In this section you'll learn how to work with SD tag data.

Chemfp supports a limited cross-toolkit API for working with SD tags. You can get a value for a single tag, the list of all tags and values, and add (and potentially replace) a tag with a given name.

NOTE: This is not a general-purpose SD tag API.

The two main goals of the SD tag API are to get a tag's value (most likely for use as an identifier) and to add a fingerprint or similarity search result to a molecule. This can be done with the toolkit's *toolkit.add_tag()* and *toolkit.get_tag()* functions:

```
>>> from chemfp import rdkit_toolkit as T
>>> mol = T.parse_molecule("O=O oxygen", "smi")
>>> T.add_tag(mol, "score", "0.9851")
>>> T.get_tag(mol, "score")
u'0.9851'
>>> print(T.create_string(mol, "sdf"))
oxygen
     RDKit

  2  1  0  0  0  0  0  0  0  0999 V2000
    0.0000    0.0000    0.0000 O   0  0  0  0  0  0  0  0  0  0  0  0
    0.0000    0.0000    0.0000 O   0  0  0  0  0  0  0  0  0  0  0  0
  1  2  2  0
M  END
> <score>
0.9851

$$$$
```

If a given tag already exists then `add_tag()` may replace the existing value, or it may add a second tag with the same name. (Eg, rdkit_toolkit currently replaces an existing tag while openeye_toolkit creates a second entry.) Chemfp does

no additional error checking, so please be careful about the use of ">" and newline characters in the tag value.

It is sometimes useful to get all of the tags and corresponding values. The toolkit's *toolkit.get_tag_pairs()* function returns these as a list of 2-element tuples, where the first term is the tag name and the second is the value:

```
>>> T.add_tag(mol, "best_id", "ABC00000123")
>>> T.add_tag(mol, "text", "This continues\nacross multiple\nlines")
>>> T.get_tag_pairs(mol)
[(u'score', u'0.9851'), (u'best_id', u'ABC00000123'), (u'text', u'This␣
↪continues\nacross multiple\nlines')]
>>> print(T.create_string(mol, "sdf"))
oxygen
     RDKit

  2  1  0  0  0  0  0  0  0  0999 V2000
    0.0000    0.0000    0.0000 O   0  0  0  0  0  0  0  0  0  0  0  0
    0.0000    0.0000    0.0000 O   0  0  0  0  0  0  0  0  0  0  0  0
  1  2  2  0
M  END
> <score>
0.9851

> <best_id>
ABC00000123

> <text>
This continues
across multiple
lines

$$$$
```

If there are multiple tags with the same name then `get_tag()` arbitrarily decides which value to return. The `get_tag_pairs()` function includes duplicates if the underlying toolkit supports it.

Add fingerprints to an SD file using a toolkit

In this section you'll learn how to add a fingerprint as a tag to the structures in an SD file using a chemistry toolkit.

The FPS and FPB fingerprint file formats store the record id and the fingerprint, but not the original structure. The most common way to tie the structure to a fingerprint is to use an SD file, and store the fingerprint as one of the tag values. (Another is to create a SMILES file variant, also called a CSV file, with the fingerprint as a new column.)

The following will parse an SD file, and for each molecule it will compute the MACCS fingerprints and add the base64-encoded fingerprint to the molecule using the unimaginative tag name "FP". It will save the results to the file named "example.sdf", which is equally unimaginative:

```
import sys
import base64
import chemfp

# Portable code to convert a fingerprint to a string
# which the underlying toolkits will accept.
#
# b64encode returns a byte string, which is fine for
# all toolkits under Python 2.
if sys.version_info.major == 2:
  b64encode = base64.b64encode
```

```
else:
  # Under Python 3, RDKit and Open Babel accept a byte string.
  # OEChem does not. Always convert to Unicode.
  def b64encode(s):
    return base64.b64encode(s).decode("ascii")

# Select your toolkit of choice
fptype = chemfp.get_fingerprint_type("OpenEye-MACCS166")
#fptype = chemfp.get_fingerprint_type("OpenBabel-MACCS")
#fptype = chemfp.get_fingerprint_type("RDKit-MACCS166")

T = fptype.toolkit

reader_args = {"rdkit.sdf.removeHs": False}
with T.read_molecules("Compound_014550001_014575000.sdf.gz",
                      reader_args=reader_args) as reader:
  with T.open_molecule_writer("example.sdf") as writer:
    for mol in reader:
      fp = fptype.compute_fingerprint(mol)
      T.add_tag(mol, "FP", b64encode(fp))
      writer.write_molecule(mol)
```

This is a very general purpose solution. It's easy to change the fingerprint type, or switch the input to a SMILES file or other supported structure file format.

(Unfortunately, there is no general purpose base64 encoder which works across all toolkits and both Python 2 and Python 3. Hence the complicated code to do the right thing.)

What it doesn't do is preserve all of the details of the input records. It converts the input record into a molecule, and back out to a new record, and the intermediate record doesn't keep all of the details.

For example, if I use OpenEye-MACCS166 and compare the first record of the original with the first record of the transformed output then the diff comparison is:

```
2c2
<   -OEChem-03291708342D
---
>   -OEChem-04301702502D
164a165,167
> > <FP>
> AACACAAAgUBgAOImoJBhQt+OKnwb
>
```

This says that the second line changed, and three new lines were added at line 164.

The second line contains a date stamp, so this isn't a big change, and the three new lines are the ones I requested. This doesn't look like much of a change, but that's because OEChem was used to make the record in the first place. Open Babel and RDKit have their own set of differences from the OEChem output defaults. For example, RDKit will sort the SD tags alphabetically.

I wanted to compare the original OEChem-based PubChem record to the output record from RDKit. I commented/uncommented the fingerprint names to use RDKit instead of OEChem. When I did this originally (since fixed), I noticed that the atom and bond counts line changed.

The first problem I noticed, before I fixed it, is that the atom and bond counts line changed. The original record has 26 atoms and bonds:

```
26 26  0     0  0  0  0  0  0999 V2000
```

while the RDKit output said there are only 15 atoms and bonds:

```
15 15  0  0  0  0  0  0  0  0999 V2000
```

What happened is that RDKit by default will convert explicit hydrogens to implicit hydrogens as part of the input process, while OEChem does not.

I can disable that in RDKit using the *removeHs* reader_arg, which is in the code I showed earlier:

```
reader_args = {"rdkit.sdf.removeHs": False}
```

With *removeHs* disabled, the RDKit atom counts match the original atom counts. There are still a few differences in the molblock.

- RDKit places a "0" in the obsolete 4th field of the counts line, while OEChem leaves it empty.
- RDKit uses the CHG property block and does not include duplicate charge information in the atom line. The PubChem file only stores charge information in the atom line.
- RDKit leaves the last three fields empty, while PubChem uses 0. These fields are respectively 'obsolete', used for "SSS queries only", and used for "Reaction, Query".

That aside, the MACCS fingerprints should be the same, right?

They are not. The RDKit (and Open Babel and CDK) MACCS keys implementations assume that all hydrogens are implicit. If there are explicit hydrogens then they will likely give a different fingerprint. If you run the above code using RDKit, with and without *removeHs*, you'll see two different values for FP:

```
AACAAAAAgUBgAKImoBBhSl/Orn0f   # RDKit, removeHs=True
AACAAAAAgUBgAKMmoJBhal/Orn0f   # RDKit, removeHs=False
```

See *MACCS dependency on hydrogens* for a more detailed description of the problem.

I'm left with the unfortunate situation where I can't preserve the explicit hydrogens without affecting the MACCS fingerprints. I think the right solution is to fix the SMARTS patterns that RDKit and others use (which is a goal of chemfp's own RDMACCS fingerprints).

Another solution for this is to use the *text_toolkit* to preserve the input SDF record syntax, and combine it with a chemistry toolkit to get the molecule you want.

Text toolkit examples

The text toolkit separates record parsing from chemical parsing. It understands the basic text structure of SDF and SMILES-based files and records, but not chemistry. It's designed with the following use cases in mind:

- add tag data to an input SDF record but keep everything else unchanged. This preserves data which might be lost by converting to then from a chemical toolkit molecule.
- synchronize operations between multiple toolkits; For example, consider a hybrid fingerprint using both OEChem and RDKit. The individual RDKit and OEChem SDF readers may get out of synch when on toolkit can't parse a record which the other can. In that case, use the text toolkit to read the records then pass the record to the chemistry toolkit.
- extract tags from an SD file. Chemfp's sdf2fps uses the text toolkit to get the id and the tag value which contains the fingerprint.

The text toolkit implements the chemistry toolkit API, except that instead of real molecule objects it uses a thin wrapper around the text for each wrapper. This chapter uses many of the concepts developed in the chapter on *Toolkit API examples*.

Toolkits may modify the molecular structure

In this section you'll learn that a chemistry toolkit might change details of a structure record so the input record and output record have some differences, even though the molecular "essence" is preserved. This is meant as an example for why you might not want to work through a chemistry toolkit molecule for everything.

The section *Add fingerprints to an SD file using a toolkit* gave an example of using a toolkit to read an SD file, compute a MACCS fingerprint, add the fingerprint as a new SD tag, and save the result to a new SD file. This is a very common task.

A problem is that toolkits can apply various normalizations, like aromaticity perception, which change atom and bond aromaticity assignments. RDKit by default will also convert explicit hydrogens into implicit hydrogens. In that section, the input record had 26 atoms and bonds while RDKit generated an output record with 15 atoms and bonds. RDKit may also 'sanitize' the structures further (for example, convert 'neutral 5 coordinate Ns with double bonds to Os to the zwitterionic form').

While it's possible to configure RDKit to keep implicit hydrogens, the RDKit MACCS fingerprinter assumes there are no explicit hydrogens. You would need to make a copy of the molecule, remove the explicit hydrogens yourself, generate the fingerprint, and then add the fingerprint to the molecule which still has the explicit hydrogens.

Bear in mind that the number of explicit atoms and bonds is based on the molecular graph model, which is only one possible representation for the actual chemical molecule. While I said there was a semantic change, the 26 atom structure and the 15 atom structure are really the same structure, just at different levels of conceptualization.

Toolkits may modify SDF syntax

In this section you'll see that passing a structure file through a chemistry toolkit and back to the same format will likely make syntax changes to the record. While not as significant as the previous section, it may help persuade you that there are cases where you want to work with the original record as text rather than as a molecule.

You will need Compound_014550001_014575000.sdf.gz from PubChem.

I'll read an SD file to get the first record as a toolkit molecule, save the molecule to SDF format, and compare the original record with the new one. This is called a round-trip test. Will there be differences?

```
import chemfp

# Select your toolkit of choice
T = chemfp.get_toolkit("openeye")
#T = chemfp.get_toolkit("rdkit")
#T = chemfp.get_toolkit("openbabel")

reader_args = {"rdkit.*.removeHs": False}
with T.read_molecules("Compound_014550001_014575000.sdf.gz",
                      reader_args=reader_args) as reader:
    with T.open_molecule_writer("example.sdf") as writer:
        for mol in reader:
            writer.write_molecule(mol)
            break  # only process the first molecule
```

If I use the "openeye" toolkit and compare its output to the first record of the input then the difference is trivial:

```
2c2
<    -OEChem-03291708342D
---
>    -OEChem-05011700162D
```

This difference is shown in the diff utility's default format. The "2c2" means there was a change in line 2, and the changed line is also on line 2. The "<" indicates the line in the first file (in this case the original PubChem file) and the ">" indicates the line in the second file (in this case "example.sdf"). The "---" is to make it easier for humans to see break between the two files.

But what does that line mean? The "CTfile" ("connection table file") spec from MDL, err, I mean Accelry, err, I mean Symyx, err, I mean BIOVIA, gives the full details. The first two characters (both blank here) are the user's initials, the next 8 characters (OpenEye uses "–" to pad out "OEChem") are the program name.

The next six character are the date, followed by 4 characters for the time. The PubChem record was created on 29 March 2017 at 08:34 while I did the transformation on 1 May 2017 at 00:16. The last two characters are the dimensionality; in this case the structure contains 2D coordinates.

PubChem used OEChem to make the file in the first place, so it's not too suprising that there weren't any differences. What about Open Babel? I changed the toolkit to "openbabel" and re-did the comparison. The first few lines of the diff were:

```
2c2
<   -OEChem-03291708342D
---
>  OpenBabel05011700222D
4c4
<  26 26  0     0  0  0  0  0  0999 V2000
---
>  26 26  0  0  0  0  0  0  0  0999 V2000
```

The 2c2 change you know already, and you can see it was a few minutes hour beween when I ran the OEChem code and the Open Babel code.

The change to line 4 is meaningless. If you look closely you'll see that OEChem has a blank in column 12 where Open Babel has a "0". The specification say that this field is obsolete, so I think you can do whatever you want there.

The rest of the differences are trivial and semantically meaningless: Open Babel uses two spaces between the ">" and "<" of a data header line, while OEChem uses one space:

```
59c59
< > <PUBCHEM_COMPOUND_CID>
---
> >  <PUBCHEM_COMPOUND_CID>
```

Finally, I'll use RDKit for the conversion. By default RDKit removes hydrogens, which would leave the result with 15 atoms. Unlike Open Babel, that action is configurable. I told RDKit to never remove hydrogens for any of its supported formats, via the reader_args:

```
reader_args = {"rdkit.*.removeHs": False}
```

(I didn't actually need the "rdkit.*." namespace prefix, but the "rdkit" helps as a reminder that this is an RDKit-specific option.)

There are the familiar changes in the second and fouth lines:

```
2c2
<   -OEChem-03291708342D
---
>      RDKit           2D
4c4
<  26 26  0     0  0  0  0  0  0999 V2000
---
>  26 26  0  0  0  0  0  0  0  0999 V2000
```

RDKit doesn't include the timestamp so leaves that fields blank. (Then again, just how useful is the timestamp? On the third hand, the chemfp fingerprint formats include a timestamp as part of the metadata, so it's odd that I question having it in another format.)

It's a bit interesting to see that RDKit changes the charge field of some of the atoms:

```
8c8
<      6.0010   -3.6550    0.0000 O   0  5  0  0  0  0  0  0  0  0  0  0
---
>      6.0010   -3.6550    0.0000 O   0  0  0  0  0  0  0  0  0  0  0  0
10c10
<      6.0010   -2.6550    0.0000 N   0  3  0  0  0  0  0  0  0  0  0  0
---
>      6.0010   -2.6550    0.0000 N   0  0  0  0  0  0  0  0  0  0  0  0
```

RDKit uses a 0 where the original used 5 (charge = -1) and 3 (charge = +1). My 2002 copy of the specification says that field is "[r]etained for compatibility with older Ctabs, M CHG and M RAD lines take precedence." I can see that the original record and the new one both contain:

```
M  CHG  2   4  -1   6   1
M  END
```

Thus, RDKit isn't as fully backwards compatible as the other two toolkits. Still, this is mostly a theoretical issue as I've not heard of someone running into a problem with how RDKit works.

Lastly, RDKit sorts the output SD tags alphabetically. I did not expect that.

While I love knowing these sorts of details, none of these (except for the explicit hydrogens) affect the semantic interpretation. Still, I can think of cases where you want to preserve the original syntax, like if you have fragile code which expects a "0" at a certain field and will crash if there's a blank.

The text toolkit "molecules"

In this section you'll learn about the molecule-like object used by the *text_toolkit*.

The text_toolkit implements the standard toolkit API, which means it reads and writes "molecules". Remember that it isn't really a chemical molecule but more like a thin layer around a molecule record. Internally these are subclasses of a *TextRecord*, though I'll often refer to them as "text molecules" to distinguish them from the the actual record as a text string.

Every text molecule has an `id` attribute, which may be None if there is no identifier, and a *record* attribute containing the actual record as a string:

```
>>> from chemfp import text_toolkit
>>> mol = text_toolkit.parse_molecule("c1ccccc1O benzene", "smi")
>>> mol
SmiRecord(id=u'benzene', record='c1ccccc1O benzene', smiles=u'c1ccccc1O',
encoding='utf8', encoding_errors='strict')
>>> mol.id   # a Unicode string
u'benzene'
>>> mol.record  # a byte string
'c1ccccc1O benzene'
>>> text_toolkit.create_string(mol, "smistring")
u'c1ccccc1O'
>>> text_toolkit.create_string(mol, "smi")
u'c1ccccc1O benzene\n'
>>> text_toolkit.create_bytes(mol, "smistring")
'c1ccccc1O'
```

```
>>> text_toolkit.create_bytes(mol, "smistring.zlib")
'x\x9cK6L\x06\x01C\x7f\x00\x0fh\x03\x04'
>>>
>>> sdf_record = (
...   'methane\n' +
...   '\n' +
...   '\n' +
...   '  1  0  0  0  0  0  0  0  0999 V2000\n' +
...   '    0.0000    0.0000    0.0000 C   0  0  0  0  0  0  0  0  0  0  0  0\n' +
...   'M  END\n' +
...   '$$$$\n')
>>>
>>> sdf_mol = text_toolkit.parse_molecule(sdf_record, "sdf")
>>> sdf_mol
SDFRecord(id_bytes='methane'(id=u'methane'), record='methane\n\n\n  1  0  0  0  0  0 ␣
↪0  0  0  0999 V2000\n    0.0 ...',
encoding='utf8', encoding_errors='strict')
>>> sdf_mol.id
u'methane'
>>> sdf_mol.record[-20:]
'0  0  0\nM  END\n$$$$\n'
```

The record is always uncompressed.

Each of the SMILES-based records has its own unique class:

```
>>> text_toolkit.parse_molecule("c1ccccc1O benzene", "smi")
SmiRecord(id=u'benzene', record='c1ccccc1O benzene', smiles=u'c1ccccc1O',
encoding='utf8', encoding_errors='strict')
>>> text_toolkit.parse_molecule("c1ccccc1O benzene", "can")
CanRecord(id=u'benzene', record='c1ccccc1O benzene', smiles=u'c1ccccc1O',
encoding='utf8', encoding_errors='strict')
>>> text_toolkit.parse_molecule("c1ccccc1O benzene", "usm")
UsmRecord(id=u'benzene', record='c1ccccc1O benzene', smiles=u'c1ccccc1O',
encoding='utf8', encoding_errors='strict')
>>> text_toolkit.parse_molecule("c1ccccc1O benzene", "smistring")
SmiStringRecord(id=None, record='c1ccccc1O', smiles=u'c1ccccc1O')
>>> text_toolkit.parse_molecule("c1ccccc1O benzene", "canstring")
CanStringRecord(id=None, record='c1ccccc1O', smiles=u'c1ccccc1O')
>>> text_toolkit.parse_molecule("c1ccccc1O benzene", "usmstring")
UsmStringRecord(id=None, record='c1ccccc1O', smiles=u'c1ccccc1O')
```

and for SMILES records you can access the SMILES directly through the *smiles* attribute:

```
>>> text_mol = text_toolkit.parse_molecule("C methane", "smistring")
>>> text_mol.smiles
u'C'
```

Each text molecule also has a *record_format* attribute, which is the format name for the record.

```
>>> text_mol.record_format
'smistring'
>>> sdf_mol.record_format
'sdf'
```

The `record_format` values are “smi”, “can”, ..., “usmstring” for the SMILES formats or “sdf” for a file in SDF format. The `record_format` will never have a compression suffix.

Unlike the chemistry-backed toolkits, the text_toolkit has no real understanding of chemistry, only a limited knowledge of the format structure. It will parse an generate garbage:

```
>>> text_mol = text_toolkit.parse_molecule("garbage", "smi")
>>> text_toolkit.create_string(text_mol, "smi", id="and trash",
...                writer_args={"delimiter": "tab"})
u'garbage\tand trash\n'
```

The `encoding` and `encoding_errors` parameters describe the character encoding of the record bytes, and how to handle errors in converting to or from that encoding. For details see the section *Unicode and other character encoding*.

The text toolkit implements the toolkit API

In this section you'll learn that the text toolkit is a pretty complete implementation of chemfp's *toolkit API*.

The text toolkit implements all of the standard toolkit API, except that it doesn't know how to convert between SMILES and SDF format. Here are some examples:

```
>>> from __future__ import print_function
>>> from chemfp import text_toolkit
>>> mol = text_toolkit.parse_molecule("C", "smistring")
>>> text_toolkit.get_id(mol) is None
True
>>> text_toolkit.set_id(mol, u"methane")
>>> text_toolkit.get_id(mol)
u'methane'
>>> text_toolkit.create_string(mol, "smi")
u'C methane\n'
>>> content = "C methane\nO=O molecular oxygen\n"
>>> with text_toolkit.read_ids_and_molecules_from_string(
...          content, "smi") as reader:
...     for id, mol in reader:
...         print("#%d %r" % (reader.location.recno, id))
...
#1 u'methane'
#2 u'molecular oxygen'
>>>
>>> writer = text_toolkit.open_molecule_writer("light.sdf")
>>> for mol in text_toolkit.read_molecules("Compound_014550001_014575000.sdf.gz"):
...   mass = text_toolkit.get_tag(mol, "PUBCHEM_EXACT_MASS")
...   mass = float(mass)
...   if mass > 100.0:
...     continue
...   cid = text_toolkit.get_tag(mol, "PUBCHEM_COMPOUND_CID")
...   print("Found", cid, mass)
...   writer.write_molecule(mol)
...
Found 14550416 77.977
Found 14550474 63.948
Found 14550599 84.021
Found 14550603 81.058
Found 14551989 85.053
Found 14556720 86.084
Found 14557343 97.039
Found 14567810 41.02
Found 14567812 57.034
```

```
Found 14567813 57.034
Found 14569188 89.097
Found 14571348 91.027
Found 14572168 86.037
Found 14572871 97.064
Found 14574249 83.032
Found 14574551 98.948
Found 14574635 50.967
Found 14574637 51.048
Found 14574638 65.064
>>> writer.close()
>>> for lineno, line in enumerate(open("light.sdf"), 1):
...     print(repr(line))
...     if lineno == 4:
...         break
...
'14550416\n'
'  -OEChem-03291708342D\n'
'\n'
'  6  5  0     0  0  0  0  0  0999 V2000\n'
```

What you can't do with the text_toolkit is convert from a SMILES-based format to SDF, or vice-versa. If you try you'll either get an exception or a meaningless molecule representation.

While you can convert between the SMILES formats, the text toolkit doesn't actually modify the SMILES term, so an input of "[238U]" will have a "canstring" (non-isomeric SMILES) of "[238U]":

```
>>> U = text_toolkit.parse_molecule("[235U]", "smistring")
>>> text_toolkit.create_string(U, "canstring")
u'[235U]'
```

I don't know if I should make this more strict in the future, and prohibit conversion between "smi", "can", and "usm" formats.

Reading and adding SD tags with the text_toolkit

In this section you'll learn how to get and set the title line and get and add tag values to an SDF record when you have the record as a block of text.

There are two ways to get or modify SD tags for an *SDFRecord*, which is the *TextRecord* subclass for files in SDF format. The first is through the standard toolkit API functions *chemfp.toolkit.get_tag()*, *chemfp.toolkit.get_tag_pairs()*, and *chemfp.toolkit.add_tag()*:

```
>>> from __future__ import print_function
>>> from chemfp import text_toolkit
>>> content = (
... "methane\n" +
... "     RDKit          \n" +
... "\n" +
... "  1  0  0  0  0  0  0  0  0999 V2000\n" +
... "    0.0000    0.0000    0.0000 C   0  0  0  0  0  0  0  0  0  0  0  0\n" +
... "M  END\n" +
... "$$$$\n")
>>> mol = text_toolkit.parse_molecule(content, "sdf")
>>> text_toolkit.add_tag(mol, "MW", "16.04246")
>>> new_record = text_toolkit.create_string(mol, "sdf")
>>> print(new_record)
```

```
methane
     RDKit

  1  0  0  0  0  0  0  0  0  0999 V2000
    0.0000    0.0000    0.0000 C   0  0  0  0  0  0  0  0  0  0  0  0
M  END
> <MW>
16.04246

$$$$

>>> new_mol = text_toolkit.parse_molecule(new_record, "sdf")
>>> text_toolkit.get_tag(new_mol, "MW")
u'16.04246'
>>> text_toolkit.get_tag_pairs(new_mol)
[(u'MW', u'16.04246')]
```

and the second is to use the corresponding methods of the text molecule: *TextRecord.get_tag()*, *TextRecord.get_tag_pairs()*, and *TextRecord.add_tag()*:

```
>>> new_mol.get_tag_pairs()
[(u'MW', u'16.04246')]
>>> new_mol.get_tag("MW")
u'16.04246'
>>>
>>> text_toolkit.get_tag_pairs(new_mol)
[(u'MW', u'16.04246')]
>>> new_mol.get_tag_pairs()
[(u'MW', u'16.04246')]
>>> new_mol.get_tag("MW")
u'16.04246'
>>> new_mol.add_tag("NUM_ATOMS", "5")
>>> print(text_toolkit.create_string(new_mol, "sdf")[-39:])
> <MW>
16.04246

> <NUM_ATOMS>
5

$$$$
```

Bear in mind that there is no way to delete a tag. This may be added in the future.

Synchronizing readers from different toolkits through the text toolkit

In this section you'll learn how to keep two different toolkit parsers synchronized by using the text toolkit to parse the records, then pass the record over to each toolkit to convert it to a molecule.

A structure file may have a couple of records which cannot be parsed by a toolkit, usually due to odd chemistry definitions. It's usually fine to skip those records, which is the purpose of the `errors="ignore"` setting. (See *Handling errors when reading molecules from a string* for more information about the *errors* parameter.)

Consider the following SMILES file with three lines:

```
% cat strange.smi
C methane
```

```
C--C ethane not for RDKit
CC ethane for everyone
```

The first and last are valid SMILES, but "C--C" is invalid. However, Open Babel will accept it, and OEChem will accept it because the default flavor does not add the "Strict" flavor flag. (See *OpenEye-specific SMILES reader_args and writer_args* for more information about OEChem flavors). As a result:

```
>>> from __future__ import print_function
>>> from chemfp import openeye_toolkit, rdkit_toolkit, openbabel_toolkit
>>> for id, mol in openeye_toolkit.read_ids_and_molecules("strange.smi", errors=
↪"ignore"):
...     print("openeye found", repr(id))
...
openeye found u'methane'
openeye found u'ethane not for RDKit'
openeye found u'ethane for everyone'
>>>
>>> for id, mol in rdkit_toolkit.read_ids_and_molecules("strange.smi", errors="ignore
↪"):
...     print("rdkit found", repr(id))
...
rdkit found u'methane'
[02:10:28] SMILES Parse Error: syntax error for input: 'C--C'
rdkit found u'ethane for everyone'
>>>
>>> for id, mol in openbabel_toolkit.read_ids_and_molecules("strange.smi", errors=
↪"ignore"):
...     print("openbabel found", repr(id))
...
openbabel found u'methane'
openbabel found u'ethane not for RDKit'
openbabel found u'ethane for everyone'
```

Sometime you want to work with multiple toolkits using the same input molecule. For example, you might want to compute a hybrid fingerprint, or make a model prediction where the descriptors come from different toolkits.

To do that, use the *text_toolkit.read_ids_and_molecules()* to read each record as a text molecule, and pass the actual record to the `toolkit.parse_molecule()` for each toolkit to get a molecule. Because I specifed the "ignore" error handler, the molecule will be None if the record could not be parsed. (See *Specify alternate error behavior* for more details.):

```
from chemfp import openeye_toolkit, rdkit_toolkit, openbabel_toolkit
from chemfp import text_toolkit

for id, text_mol in text_toolkit.read_ids_and_molecules("strange.smi", errors="ignore
↪"):
    if openeye_toolkit.parse_molecule(text_mol.record, text_mol.record_format ,␣
↪errors="ignore"):
        print("openeye parsed", repr(id))
    else:
        print("openeye could not parse", repr(id))

    if rdkit_toolkit.parse_molecule(text_mol.record, text_mol.record_format , errors=
↪"ignore"):
        print("rdkit parsed", repr(id))
    else:
        print("rdkit could not parse", repr(id))
```

```
    if openbabel_toolkit.parse_molecule(text_mol.record, text_mol.record_format ,␣
↪errors="ignore"):
        print("openbabel parsed", repr(id))
    else:
        print("openbabel could not parse", repr(id))
```

The output from running the above is:

```
openeye parsed 'methane'
rdkit parsed 'methane'
openbabel parsed 'methane'
openeye parsed 'ethane not for RDKit'
[03:23:38] SMILES Parse Error: syntax error for input: C--C
rdkit could not parse 'ethane not for RDKit'
openbabel parsed 'ethane not for RDKit'
openeye parsed 'ethane for everyone'
rdkit parsed 'ethane for everyone'
openbabel parsed 'ethane for everyone'
```

The above works, but there's a lot of duplicate code, I don't like the layout for the output, and there's bit of extra overhead to re-interpret the `parse_molecule()` for each call. I'll make a space-delimited file as output, and use *`toolkit.make_id_and_molecule_parser()`* to create a specialized parser for each available toolkit:

```
from __future__ import print_function
import chemfp
from chemfp import text_toolkit

reader = text_toolkit.read_ids_and_molecules("strange.smi")
format = reader.metadata.record_format

column_headers = []
parsers = []
for toolkit_name in ("openeye", "rdkit", "openbabel"):
  column_headers.append(toolkit_name)
  try:
    toolkit = chemfp.get_toolkit(toolkit_name)
  except ValueError:
    parsers.append(None)
  else:
    parser = toolkit.make_id_and_molecule_parser(format, errors="ignore")
    parsers.append(parser)

column_headers.append("ID")

print(*column_headers, sep="\t")  # print the header

for id, text_mol in reader:
  columns = []
  for parser in parsers:
    if parser is None:
      columns.append("N/A")
    else:
      id, mol = parser(text_mol.record)
      if mol is not None:
        columns.append("Yes")
      else:
        columns.append("No")
  columns.append(id)
```

```
  print(*columns, sep="\t")
```

This writes a tab-delimited file to stdout, ready for import into any spreadsheet program:

```
openeye       rdkit   openbabel       ID
Yes   Yes     Yes     methane
Yes   No      Yes     ethane not for RDKit
Yes   Yes     Yes     ethane for everyone
```

(There will also be an error message from RDKit sent to stderr.)

Add multiple toolkit fingerprints to an SD file

In this section you'll learn how to use multiple toolkits to generate fingerprints for each molecule in an SD file, and add the fingerprints results back to the record as new SD tags.

In *Add fingerprints to an SD file using a toolkit* you learned how to use a toolkit to read a file as molecules, compute a fingerprint for each molecule, and add the fingerprint to the molecule as an SD tag, and save the result to a new SD file. The processing pipeline converted the input to a toolkit molecule and out again, and in doing so changed other parts of the record besides the new SD tag.

Sometimes you want to preserve the input as much as you can. For that case you can use the text reader to get text molecules, pass each text molecule's record that to the toolkit to compute the fingerprint, add the new fingerprint as a tag for the text molecule, and save the result to a file.

I'll do that one better; I'll generate fingerprints using multiple toolkit and add all of them to the output file. Here's an example of what the end of a new record will look like. Note: although the fingerprints are actually on one line, I've folded the long fingerprints across multiple lines so it doesn't overflow this page.

```
> <PUBCHEM_BONDANNOTATIONS>
18  20  8
20  21  8
6  18  8
6  8  8
8  21  8

> <rdkit512>
fdcef73b7efeddff9fc5fbfffbf7ff7fddf9ff7fc43d7fa63e853ee3f77bfdbdb5ff
fffbfd67c6b9f3bff1febf7dff9ffefdbf1d3fffeb7f2fff7fe53bfffbff

> <rdkit1024>
7dcc333a76bec9fb1f05fbc57bf4ff5e8579f65fc02c78023a051ea21739d835b15c
96f17d20c2b9708ed1580b24ca9afe55ae1d1dff2a7827f667e021a9e9d68482c601
5ac014e581c45a7adbe74e235c903f78c4154fa6268534c3f15bbd8d95a7ffbbbd47
8611c33ff0f6b67d7d158efd9918265ce15f09ef3a451b7ebabb

> <obfp3>
0400000000b001

$$$$
```

I'll break it down into stages. The first is some preamble code to import the modules and configure the input and output files:

```
import chemfp

# I'll use chemfp's text-based SD parser, so the output SD records
```

```
# will be identical to the input records, except to append the new
# tags at the end of each record.

from chemfp import text_toolkit

input_filename = "/Users/dalke/databases/pubchem/Compound_001000001_001025000.sdf.gz"
output_filename = "output.sdf.gz"
```

Next is to get the right *SDF parser* (a function which converts an SDF record into a identifier and a native toolkit molecule) and *fingerprinter* (a function which converts a toolkit molecule into a fingerprint) for each fingerprint type.

```
# The list of tag names and the corresponding fingerprint types.
wanted_fingerprint_types = (
    ("rdkit512", "RDKit-Fingerprint fpSize=512"),
    ("rdkit1024", "RDKit-Fingerprint fpSize=1024"),
    ("obfp3", "OpenBabel-FP3"),
    )

build_data = []   # I'll use this to build the fingerprint data.
toolkit_sdf_record_parsers = {}  # I'll use this to convert an SD record into a␣
↪molecule.

for output_tag, fingerprint_type_string in wanted_fingerprint_types:
    # First, get the corresponding fingerprint type.
    fingerprint_type = chemfp.get_fingerprint_type(fingerprint_type_string)

    # Figure out which toolkit to use to parse the SD records.
    toolkit = fingerprint_type.toolkit

    # For each unique toolkit, get a function that turns an SD record into a molecule.
    # (If multiple fingerprints use the same toolkit then I only
    # need to parse it once.)
    if toolkit.name not in toolkit_sdf_record_parsers:
        # The "ignore" means to return None on error, rather than raise an exception.
        toolkit_sdf_record_parsers[toolkit.name] = toolkit.make_id_and_molecule_
↪parser("sdf", errors="ignore")

    # Get a function which turns a molecule into a fingerprint.
    fingerprinter = fingerprint_type.make_fingerprinter()

    # Store this information for record processing.
    build_data.append( (output_tag, toolkit.name, fingerprinter) )
```

Finally, use the text toolkit to read text molecules for each record, then use the SDF parser to get the id and molecule from the record text, then the fingerprinter to get the fingerprint from the molecule:

```
# Use the text toolkit to read and write SDF records.
with text_toolkit.open_molecule_writer(output_filename) as writer:
    for text_mol in text_toolkit.read_molecules(input_filename):

        # The text "molecule" .record is the actual text.
        record = text_mol.record

        # Make the fingerprints for each record and append the tag.

        # For extra performance, cache parsed molecules for future use.
        toolkit_mols = {}
```

```
        for output_tag, toolkit_name, fingerprinter in build_data:
            # There's no need to reparse the record if I've seen it before.
            if toolkit_name in toolkit_mols:
                toolkit_mol = toolkit_mols[toolkit_name]
            else:
                # Parse the record and save the molecule for later.
                toolkit_id, toolkit_mol = toolkit_sdf_record_parsers[toolkit_
↪name](record)
                toolkit_mols[toolkit_name] = toolkit_mol

            if toolkit_mol is None:
                # There's no molecule, so no fingerprint. Save the empty string.
                text_mol.add_tag(output_tag, "")
            else:
                # Make a fingeprint and save it to the tag as a hex-encoded string.
                fp = fingerprinter(toolkit_mol)
                text_mol.add_tag(output_tag, fp.encode("hex"))

        # Write the text molecule to the output stream.
        writer.write_molecule(text_mol)
```

Text toolkit and SDF files

In this section you'll learn about the specialized SDF reader API to read SDF records and tag values directly instead of through a text record.

The text toolkit support for the toolkit API lets you use the same code for SDF and SMILES, and switch between text-based and molecule-based parsers. Genericness comes at a cost. The *TextRecord* class is a wrapper around the actual record, so at the least there is some overhead for creating a wrapper for each record.

The text toolkit has special support for reading SDF records as raw byte strings, which are not wrapped in any object. There several SDF reader variations depending on if you want to read from a file or a string, and if you want to read the record, the (id, record) pair, or an (id, tag value) pair. These functions are:

- *read_sdf_records()* - iterate over the records in an SD file
- *read_sdf_records_from_string()* - the same, but from a string
- *read_sdf_ids_and_records()* - iterate over the (id, record string) pairs from an SD file
- *read_sdf_ids_and_records_from_string()* - the same, but from a string
- *read_sdf_ids_and_values()* - iterator over the (id, value) pairs from an SD file
- *read_sdf_ids_and_values_from_string()* - the same, but from a string

(Note: while I write this as (id, value), those are just labels. By default it returns (SD title, SD title) pairs, or you can specify an alternate *id_tag* and *value_tag* to get the pairs you want.)

There are also special functions to work with the tag data and title of an SDF record, which take the record string as input:

- *get_sdf_tag()* - get a named tag from an SDF record
- *add_sdf_tag()* - return a new SDF record with the new tag and value at the end of the tag block
- *get_sdf_tag_pairs()* - return a list of (tag name, tag value) pairs
- *get_sdf_id()* - return the first line of the SDF record

- *set_sdf_id()* - return a new SDF record with the new title line

The next few sections will cover some examples of how to use these specialized functions.

Read id and tag value pairs from an SD file

In this section you'll learn how read the (id, tag value) for each record in an SD file using a specialized SDF reader. You will need Compound_014550001_014575000.sdf.gz from PubChem.

The specialized SDF readers are faster than the more generic text_toolkit support for the toolkit API. As an example, I'll extract the identifer and molecular weight field from a PubChem file using the (slower) chemfp toolkit API:

```
from __future__ import print_function
from chemfp import text_toolkit

filename = "Compound_014550001_014575000.sdf.gz"
with text_toolkit.read_ids_and_molecules(filename) as reader:
  for id, text_mol in reader:
    mw = text_mol.get_tag("PUBCHEM_EXACT_MASS")
    print(id, mw)
```

Next I'll extract it using the (faster) *read_sdf_ids_and_values()* function, which returns an iterator of the (id, tag value) pairs. Just like with *toolkit.read_ids_and_molecules()*, by default the id is the title line of the SD record, or I can use the *id_tag* parameter to get it from one of the SD tags. The *value_tag* has the same meaning; by default the value is the record's title, or I can specify an alternate tag name containing the value to use:

```
from __future__ import print_function
from chemfp import text_toolkit

filename = "Compound_014550001_014575000.sdf.gz"
with text_toolkit.read_sdf_ids_and_values(filename, value_tag="PUBCHEM_EXACT_MASS")␣
↪as reader:
  for id, mw in reader:
    print(id, mw)
```

Both of these generate output starting with:

```
14550001 229.041
14550002 199.067
14550003 169.056
14550004 124.1
14550005 291.954
14550010 259.061
14550011 224.05
```

My timings show that the first, generic implementation takes 0.26 seconds while the second, specialized implementation takes 0.17 seconds, which is about 33% faster, or enough to save about an hour when parsing PubChem. (The difference is even larger without the gzip overhead.) That's why the sdf2fps command-line tool uses this function to extract the ids and fingerprint values from PubChem files.

Extract the id and atom and bond counts from an SD file

In this section you'll use a specialized SDF reader iterate over the records of an SD file. You will need Compound_014550001_014575000.sdf.gz from PubChem.

The "records" returned by *read_sdf_records()*, *read_sdf_records_from_string()*, *read_sdf_ids_and_records()*, and *read_sdf_ids_and_records_from_string()* are the actual record content as a string, and not wrapped in a TextRecord or other class.

For example, the following will read each record from an SD file and use a regular expression to extract the title line, the number of atoms from the first 3 characters of line 4, and the number of bonds as the second 3 characters of line 4:

```
from __future__ import print_function
from chemfp import text_toolkit
import re

pat = re.compile(br"(.*)\n.*\n.*\n(...)(...)")

filename = "Compound_014550001_014575000.sdf.gz"
for record in text_toolkit.read_sdf_records(filename):
  m = pat.match(record)
  id = m.group(1).decode("utf8")
  num_atoms = int(m.group(2))
  num_bonds = int(m.group(3))
  print(id, num_atoms, num_bonds)
```

The output starts:

```
14550001 26 26
14550002 26 26
14550003 22 22
14550004 21 21
14550005 24 23
14550010 31 31
14550011 27 28
14550044 166 162
```

(Bear in mind that there may also be implicit hydrogens, so unless you know that all hydrogens are explicit or implicit, these numbers may only be roughly useful.)

Records are byte strings

The example code, while short, is still a bit tricky. The reader returns the SD records as byte strings, not Unicode strings. Why? First and foremost, using Python to read bytes from a file is 2-3x faster than reading Unicode. If all you care about is reading a couple of fields from the record then it's faster to work with bytes and convert only those fields.

Second, this is a low-level API meant to give the actual byte representation of the data. Among other things, you should be able to know exactly where the record is located in the file. You can even do things like handle mixed encodings, where one tag value is UTF-8 encoded and another is Latin-1 encoded and cannot be read as a value UTF-8.

Python 3 makes a strong distinction between a byte string and a Unicode string. For Python 3, because the record a byte string, you'll have to use a byte-based regular expression to parse it, as in:

```
pat = re.compile(br"(.*)\n.*\n.*\n(...)(...)")
```

You'll also have to convert the title bytes to Unicode if you want to print the result, as in:

```
id = m.group(1).decode("utf8")
```

Thankfully, int() knows how to read the ASCII digits from a byte string, so I didn't have to do extra work there.

SDF-specific parser parameters

In this section you'll learn that the specialized SDF readers support the standard *errors* and *location* , and have a few special parameters of their own. You will need Compound_014550001_014575000.sdf.gz from PubChem.

All six of the `read_sdf_*` functions support the same *errors* and *location* parameters as the standard toolkit API, with the same meaning. For example, the following shows where each record is located in the uncompressed file:

```
from __future__ import print_function
from chemfp import text_toolkit

filename = "Compound_014550001_014575000.sdf.gz"
with text_toolkit.read_sdf_ids_and_records(filename) as reader:
  loc = reader.location
  for id, record in reader:
    start_byte, end_byte = loc.offsets
    print("%s at line %d (bytes %d-%d)" % (id, loc.lineno, start_byte, end_byte))
```

The output starts:

```
14550001 at line 1 (bytes 0-4227)
14550002 at line 166 (bytes 4227-8399)
14550003 at line 330 (bytes 8399-12140)
14550004 at line 486 (bytes 12140-15744)
14550005 at line 639 (bytes 15744-19838)
```

See *Handling errors when reading molecules from a string* for more information about the *errors* parameter, and *Location information: record position and content* for a description of the how to use a `Location` to the record's first line number and start/end offsets in the file.

The six functions do not have a *format* option, because the format must be "sdf" or "sdf.gz". Instead, there is a *compression* parameter. The default of None selects the compression type based on the filename, if the filename is available, or assumes the input is uncompressed. Use "gz" if the input is gzip'ed, and "none" or "" if the input is uncompressed.

The *block_size* is a tunable parameter, with a default value of 320 KB. The underlying reader reads a block of text then tries to extract records. When it gets to the end of a block, it reads a new block, and prepends the remaining part of the old block to the new one before looking for new records.

For performance reasons, the *block_size* should be several times larger than the largest record. During error recovery, the reader will read up to 320 KB or `5*block_size`, whichever is larger, in order to find the next "$$$$" line and resynchronize.

Working with SD records as strings

In this section you'll learn about the helper functions to work with SD record id and tag data when the SD record is a string. You will need Compound_014550001_014575000.sdf.gz from PubChem.

I'll use one of the specialized SD file readers, `read_sdf_records()`, to get the first record from an SD file:

```
>>> from __future__ import print_function
>>> from chemfp import text_toolkit
>>> record = next(text_toolkit.read_sdf_records("Compound_014550001_014575000.sdf.gz
↪"))
>>> print(record[:100])
14550001
  -OEChem-03291708342D

```

```
 26 26  0     0  0  0  0  0  0999 V2000
    5.1350    0.8450    0.0
```

I can use *get_sdf_tag()* and *get_sdf_tag_pairs()* to get information about the tags in the record:

```
>>> for tag_name, tag_value in text_toolkit.get_sdf_tag_pairs(record):
...   print(tag_name, "=", repr(tag_value[:40]))
...
PUBCHEM_COMPOUND_CID = '14550001'
PUBCHEM_COMPOUND_CANONICALIZED = '1'
PUBCHEM_CACTVS_COMPLEXITY = '209'
PUBCHEM_CACTVS_HBOND_ACCEPTOR = '5'
PUBCHEM_CACTVS_HBOND_DONOR = '2'
PUBCHEM_CACTVS_ROTATABLE_BOND = '4'
PUBCHEM_CACTVS_SUBSKEYS = 'AAADccByOABAAAAAAAAAAAAAAAAAAAAAAAAAwAAAA'
PUBCHEM_IUPAC_OPENEYE_NAME = '2-(2-hydroxyethylsulfanylmethyl)-4-nitro'
          ...
>>> text_toolkit.get_sdf_tag(record, "PUBCHEM_IUPAC_OPENEYE_NAME")
u'2-(2-hydroxyethylsulfanylmethyl)-4-nitro-phenol'
```

or use *add_sdf_tag()* to create a new record with a given tag and value added to the end of the tag block:

```
>>> print(record[-90:])
> <PUBCHEM_BONDANNOTATIONS>
10  13  8
11  14  8
13  14  8
7  10  8
7  9  8
9  11  8

$$$$

>>> new_record = text_toolkit.add_sdf_tag(record, b"VOLUME", b"123.45")
>>> print(new_record[-109:])
> <PUBCHEM_BONDANNOTATIONS>
10  13  8
11  14  8
13  14  8
7  10  8
7  9  8
9  11  8

> <VOLUME>
123.45

$$$$
```

I can also get the title line of the SD record using *get_sdf_id()*:

```
>>> text_toolkit.get_sdf_id(record)
'14550001'
```

or create a new string which is the old string with the title line replaced by a new value:

```
>>> new_record = text_toolkit.set_sdf_id(record, b"987ZYX")
>>> text_toolkit.get_sdf_id(new_record)
'987ZYX'
```

Note that I used byte strings, like b"VOLUME" and b"987ZYX". In general the values must be of the same string type as the record. On the flip side, if you have a Unicode record then you must pass in Unicode strings as values:

```
>>> unicode_record = record.decode("utf8")
>>> new_record = text_toolkit.set_sdf_id(unicode_record, u"Hello")
>>> new_record[:6]
u'Hello\n'
```

Unicode and other character encoding

In this section you'll learn a bit about how the text toolkit deals with different character encodings. This is a hard topic and I won't cover it in full details. If you have a problem with Unicode encodings (and hopefuly a support contract) then contact me and I'll help that way.

The SDF format is 8-bit clean. The specification itself uses ASCII but fields like the title, the tag name, and the tag value can contain nearly any byte value. (Some values like newline and '<' and '>' in the tag name, have special meaning and must not be used.)

Unfortunately, different software handle those non-ASCII values differently. An older Unix system might use the Latin-1 character set, which is able to handle many European and some non-European languages, but doesn't have the Euro currency symbol. Microsoft Windows code page 1252 is effectively a superset of Latin-1, with the Euro symbol and a several other additional symbols.

There are of course many other symbols. The consensus for new systems is to use UTF-8 encoded Unicode, which is compatible with 8-bit clean ASCII and can handle most of the world's languages, plus a large number of symbols. This encoding may use one, two, or more bytes to represent each symbol.

The Python3 bindings of OpenEye, RDKit, and Open Babel's have all decided to interpret SD files as UTF-8 encoded. This consensus is great ... so long as your files are also compatible with UTF-8. But what if they aren't? What if you have to read Latin-1 encoded file, or worse, a file where different fields have multiple encodings?

To demonstrate the problem, I'll construct a problematic file for β-methylphenethylamine, with an experimental melting point of 140-142°C, stored in a Latin-1 encoded SD file. For now I'll use use 'Beta' for the name, and 'DEGREE' for the temperature, as placeholders for the two non-ASCII characters.

```
>>> from __future__ import print_function
>>> from chemfp import rdkit_toolkit as T # use your toolkit of choice
>>> mol = T.parse_molecule("NCC(c1ccccc1)C", "smi")
>>> T.set_id(mol, "Beta-methylphenethylamine")
>>> T.add_tag(mol, "MP", "140-142DEGREEC")
>>> unicode_record = T.create_string(mol, "sdf")
>>> print(unicode_record)
Beta-methylphenethylamine
     RDKit

 10 10  0  0  0  0  0  0  0  0999 V2000
    0.0000    0.0000    0.0000 N   0  0  0  0  0  0  0  0  0  0  0  0
    0.0000    0.0000    0.0000 C   0  0  0  0  0  0  0  0  0  0  0  0
    0.0000    0.0000    0.0000 C   0  0  0  0  0  0  0  0  0  0  0  0
    0.0000    0.0000    0.0000 C   0  0  0  0  0  0  0  0  0  0  0  0
    0.0000    0.0000    0.0000 C   0  0  0  0  0  0  0  0  0  0  0  0
    0.0000    0.0000    0.0000 C   0  0  0  0  0  0  0  0  0  0  0  0
    0.0000    0.0000    0.0000 C   0  0  0  0  0  0  0  0  0  0  0  0
    0.0000    0.0000    0.0000 C   0  0  0  0  0  0  0  0  0  0  0  0
    0.0000    0.0000    0.0000 C   0  0  0  0  0  0  0  0  0  0  0  0
    0.0000    0.0000    0.0000 C   0  0  0  0  0  0  0  0  0  0  0  0
  1  2  1  0
```

```
  2  3  1  0
  3  4  1  0
  4  5  2  0
  5  6  1  0
  6  7  2  0
  7  8  1  0
  8  9  2  0
  3 10  1  0
  9  4  1  0
M  END
> <MP>
140-142DEGREEC

$$$$
```

Next, I'll replace the 'DEGREE' with the corresponding Unicode characters. (I'll use the long Unicode name to be explicit.)

```
>>> unicode_record = unicode_record.replace(u"DEGREE", u"\N{DEGREE SIGN}")
>>> print(unicode_record)
Beta-methylphenethylamine
     RDKit

 10 10  0  0  0  0  0  0  0  0999 V2000
     ....
M  END
> <MP>
140-142°C

$$$$
```

Finally, I'll save it to the file "latin1.sdf", using the Latin-1 encoding:

```
>>> open("latin1.sdf", "wb").write(unicode_record.encode("latin1"))
```

This is not valid UTF-8. In my terminal, the MP tag value looks like:

```
> <MP>
140-142C
```

where the "" is the special symbol for REPLACEMENT CHARACTER, meaning that the actual character cannot be shown.

What happens if I read the file using each of the native toolkit APIs? First, OEChem under both Python 2.7 and Python 3.6:

```
>>> from openeye.oechem import *
>>> ifs = oemolistream("latin1.sdf")
>>> mol = OEGraphMol()
>>> OEReadMolecule(ifs, mol)
True
>>> OEGetSDData(mol, "MP")  # OEChem on Python 2.7
'140-142\xb0C'

>>> OEGetSDData(mol, "MP")  # OEChem on Python 3.6
'140-142\udcb0C'
```

Remember, OEGetSDData() on Python 2.7 returns byte strings, and you'll need to decode that string manually to get

the degree symbol. While OEGetSDData() on Python 3.5 returns Unicode strings, but the byte "\xb0" is not a valid UTF-8 encoding. Instead, OEChem uses the Unicode codepoint "\udcb0". This is a surrogate for the actual character, and something I don't fully understand. Various sources say this is a UTF-16 behavior which isn't correct UTF-8. Python doesn't like it:

```
>>> print('140-142\udcb0C')
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
UnicodeEncodeError: 'utf-8' codec can't encode character '\udcb0' in position 7:␣
↪surrogates not allowed
```

Next, Open Babel under both Python 2.7 and Python 3.6:

```
>>> import openbabel as ob
>>> conv = ob.OBConversion()
>>> mol = ob.OBMol()
>>> conv.ReadFile(mol, "latin1.sdf")
True
>>> mol.GetData("MP").GetValue()  # Open Babel on Python 2.7
'140-142\xb0C'

>>> mol.GetData("MP").GetValue()  # Open Babel on Python 3.6
'140-142\udcb0C'
```

Open Babel gives exactly the same results as OEChem.

Finally, RDKit:

```
>>> from rdkit import Chem
>>> supplier = Chem.ForwardSDMolSupplier("latin1.sdf")
>>> mol = next(supplier)
>>> mol.GetProp("MP")       # RDKit on Python 2.7
'140-142\xb0C'

>>> mol.GetProp("MP")       # RDKit on Python 3.6
'140-142\xb0C'
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
UnicodeDecodeError: 'utf-8' codec can't decode byte 0xb0 in position 7: invalid start␣
↪byte
```

RDKit doesn't give a surrogate value for the illegal UTF-8 character. Instead, it complains. Which also means there is no way to get that data from Python.

What do you do if you have to read a Latin-1 encoded SD file? One solution is to use an external tool like iconv to translate the file to UTF8.

```
% iconv -f latin1 -t utf-8 < latin1.sdf > utf8.sdf
```

Another is to use Python to convert the entire file from Latin-1 to UTF8 then pass the transcoded contents to the toolkit:

```
>>> content = open("latin1.sdf", "rb").read()
>>> content = content.decode("latin1").encode("utf8")
>>>
>>> from __future__ import print_function
>>> import chemfp
>>> for tk in ("openbabel", "openeye", "rdkit"):
...   T = chemfp.get_toolkit(tk)
```

```
...   for mol in T.read_molecules_from_string(content, "sdf"):
...     print(tk, T.get_tag(mol, "MP"))
openbabel 140-142°C
openeye 140-142°C
rdkit 140-142°C
```

But if all you want is some of the tag data values, and not the molecule, then you can ask the text_toolkit to read the record as a "latin1" encoded file:

```
>>> from chemfp import text_toolkit
>>> for mol in text_toolkit.read_molecules("latin1.sdf", encoding="latin1"):
...   print(mol.get_tag("MP"))
...
140-142°C
```

The content is converted on-demand, that is, only when `get_id()` or `get_tag()` are called. The text_toolkit's "molecule" stores the encoding so it knows how to decode the fields:

```
>>> mol.encoding
'latin1'
```

By the way, if you omit the 'encoding="latin1"' parameter then you'll get an exception:

```
>>> for mol in text_toolkit.read_molecules("latin1.sdf"):
...   print(mol.get_tag("MP"))
...
Traceback (most recent call last):
  File "<stdin>", line 2, in <module>
  File "/Users/dalke/cvses/cfp-3x/docs/tmp/chemfp/_text_toolkit.py", line 258, in get_
↪tag
    return get_sdf_record_tag(self.record, tag, self.encoding, self.encoding_errors)
  File "/Users/dalke/cvses/cfp-3x/docs/tmp/chemfp/_text_toolkit.py", line 1474, in␣
↪get_sdf_record_tag
    return value.decode(encoding, encoding_errors)
UnicodeDecodeError: 'utf-8' codec can't decode byte 0xb0 in position 7: invalid start␣
↪byte
```

Mixed encodings and raw bytes

In this section you'll learn how to get access to the id and tag data as byte strings rather than Unicode strings. This might be used if you have a perverse file which uses multiple encodings. If you run into that case, let me know - I'll give you a sympathy prize for having to deal with it.

In the previous section you learned a few ways to read an Latin-1 encoded SD file. What happens if the title line contains an id which is UTF-8 encoded while the tag data contains a Latin-1 encoded value? (Or if you have to deal with a 'clever' programmer who put in semi-binary data into a data field. Because that's the sort of thing we clever programmers sometimes do.)

The techniques I mentioned in that previous section won't work because they assume the entire file has the same encoding.

Instead, use the text_toolkit to read the file, but access it through the byte API rather than the string API.

I need an example file. I'll start with the "latin1.sdf" file I created for the previous section, which uses a Latin-1 encoded degree symbol in the "MP" tag data. I'll modify it so the "Beta" in the title line is replaced by the UTF-8 encoded "β" character.

```
>>> content = open("latin1.sdf", "rb").read()
>>> mixed_content = content.replace(b"Beta", u"\N{GREEK SMALL LETTER BETA}".encode(
↪"utf8"))
>>> open("mixed.sdf", "wb").write(mixed_content)
```

(On Python 3, that last line will return "946", to indicate that it wrote 946 bytes to the file.)

On a UTF-8 terminal the title line and the MP tag data line are respectively:

On a Latin-1 terminal they are:

How do I get their "real" values? I'll use the text_toolkit to read the first record from the file:

```
>>> from chemfp import text_toolkit
>>> mol = next(text_toolkit.read_molecules("mixed.sdf"))
>>> mol
SDFRecord(id_bytes='\xce\xb2-methylphenethylamine'(id=u'\u03b2-methylphenethylamine'),
record='\xce\xb2-methylphenethylamine\n     RDKit          \n\n 10 10  0    ...',
encoding='utf8', encoding_errors='strict')
```

The title line is in utf8 so that's not a problem

```
>>> print(mol.id)
β-methylphenethylamine
```

But I won't be able to read the "MP" field because it's not UTF-8 encoded:

```
>>> mol.get_tag("MP")
Traceback (most recent call last):
  ...
UnicodeDecodeError: 'utf8' codec can't decode byte 0xb0 in position 7: invalid start␣
↪byte
```

Instead, I'll use *get_tag_as_bytes()* to get the underlying bytes for the named tag, rather than as converted to a Unicode string:

```
>>> mol.get_tag_as_bytes(b"MP")
'140-142\xb0C'
```

Once I have the bytes, I can decode them as Latin-1:

```
>>> print(mol.get_tag_as_bytes(b"MP").decode("latin1"))
140-142°C
```

Note that this function requires the tag name be the byte string which is found in the file. A Unicode name (which is the default string type under Python 3) will raise an exception:

```
>>> mol.get_tag_as_bytes(u"MP")
Traceback (most recent call last):
   ...
ValueError: tag must be a byte string or None
```

Use method *get_tag_pairs_as_bytes()* to get the list of all (tag, data) pairs, where both the tag and data are return as byte strings.

```
>>> mol.get_tag_pairs_as_bytes()
[('MP', '140-142\xb0C')]
```

Finally, use `id_bytes` to get the raw bytes for the identifier:

```
>>> mol.id_bytes
'\xce\xb2-methylphenethylamine'
```

For example, if I read the file as Latin-1 then the Unicode `id` "MP" tag be what I expected, the id won't be correct. Instead, I can get the `id_bytes` and decode it manually as UTF-8:

```
>>> mol2 = next(text_toolkit.read_molecules("mixed.sdf", encoding="latin1"))
>>> print(mol2.get_tag("MP"))
140-142°C
>>> mol2.id
u'\xce\xb2-methylphenethylamine'
>>> print(mol2.id)
Î²-methylphenethylamine
>>>
>>> print(mol2.id_bytes.decode("utf8"))
β-methylphenethylamine
```

chemfp API

This chapter contains the docstrings for the public portion of the chemfp API.

chemfp top-level module

The following functions and classes are in the top-level chemfp module.

chemfp.**open**(*source*, *format=None*, *location=None*)

Read fingerprints from a fingerprint file

Read fingerprints from *source*, using the given format. If *source* is a string then it is treated as a filename. If *source* is None then fingerprints are read from stdin. Otherwise, *source* must be a Python file object supporting the `read` and `readline` methods.

If *format* is None then the fingerprint file format and compression type are derived from the source filename, or from the `name` attribute of the source file object. If the source is None then the stdin is assumed to be uncompressed data in "fps" format.

The supported format strings are:

- "fps", "fps.gz" for fingerprints in FPS format
- "fpb" for fingerprints in FPB format

The optional *location* is a *chemfp.io.Location* instance. It will only be used if the source is in FPS format.

If the source is in FPS format then `open` will return a *chemfp.fps_io.FPSReader*, which will use the *location* if specified.

If the source is in FPB format then `open` will return a *chemfp.arena.FingerprintArena* and the *location* will not be used.

Here's an example of printing the contents of the file:

```
from chemfp.bitops import hex_encode
reader = chemfp.open("example.fps.gz")
```

```
for id, fp in reader:
    print(id, hex_encode(fp))
```

Parameters

- **source** (*A filename string, a file object, or None*) – The fingerprint source.
- **format** (*string, or None*) – The file format and optional compression.

Returns a *chemfp.fps_io.FPSReader* or *chemfp.arena.FingerprintArena*

chemfp.**load_fingerprints**(*reader*, *metadata=None*, *reorder=True*, *alignment=None*, *format=None*)

Load all of the fingerprints into an in-memory FingerprintArena data structure

The function reads all of the fingerprints and identifers from *reader* and stores them into an in-memory *chemfp.arena.FingerprintArena* data structure which supports fast similarity searches.

If *reader* is a string or has a `read` attribute then it will be passed to the *chemfp.open()* function and the result used as the reader. If that returns a FingerprintArena then the *reorder* and *alignment* parameters are ignored and the arena returned.

If *reader* is a FingerprintArena then the *reorder* and *alignment* parameters are ignored. If *metadata* is None then the input reader is returned without modifications, otherwise a new FingerprintArena is created, whose metadata attribue is *metadata*.

Otherwise the *reader* or the result of opening the file must be an iterator which returns (id, fingerprint) pairs. These will be used to create a new arena.

metadata specifies the metadata for all returned arenas. If not given the default comes from the source file or from `reader.metadata`.

The loader may reorder the fingerprints for better search performance. To prevent ordering, use `reorder=False`. The *reorder* parameter is ignored if the reader is an arena or FPB file.

The *alignment* option specifies the alignment data alignment and padding size for each fingerprint. A value of 8 means that each fingerprint will start on a 8 byte alignment, and use storage space which a multiple of 8 bytes long. The default value of None will determine the best alignment based on the fingerprint size and available popcount methods. This parameter is ignored if the reader is an arena or FPB file.

Parameters

- **reader** (*a string, file object, or (id, fingerprint) iterator*) – An iterator over (id, fingerprint) pairs
- **metadata** (Metadata) – The metadata for the arena, if other than reader.metadata
- **reorder** (*True or False*) – Specify if fingerprints should be reordered for better performance
- **alignment** (*a positive integer, or None*) – Alignment size in bytes (both data alignment and padding); None autoselects the best alignment.
- **format** (*None, "fps", "fps.gz", or "fpb"*) – The file format name if the reader is a string

Returns *chemfp.arena.FingerprintArena*

chemfp.**read_molecule_fingerprints**(*type*, *source=None*, *format=None*, *id_tag=None*, *reader_args=None*, *errors="strict"*)

Read structures from *source* and return the corresponding ids and fingerprints

This returns an *chemfp.fps_io.FPSReader* which can be iterated over to get the id and fingerprint for each read structure record. The fingerprint generated depends on the value of *type*. Structures are read from *source*, which can either be the structure filename, or None to read from stdin.

type contains the information about how to turn a structure into a fingerprint. It can be a string or a metadata instance. String values look like `OpenBabel-FP2/1`, `OpenEye-Path`, and `OpenEye-Path/1 min_bonds=0 max_bonds=5 atype=DefaultAtom btype=DefaultBond`. Default values are used for unspecified parameters. Use a Metadata instance with *type* and *aromaticity* values set in order to pass aromaticity information to OpenEye.

If *format* is None then the structure file format and compression are determined by the filename's extension(s), defaulting to uncompressed SMILES if that is not possible. Otherwise *format* may be "smi" or "sdf" optionally followed by ".gz" or ".bz2" to indicate compression. The OpenBabel and OpenEye toolkits also support additional formats.

If *id_tag* is None, then the record id is based on the title field for the given format. If the input format is "sdf" then *id_tag* specifies the tag field containing the identifier. (Only the first line is used for multi-line values.) For example, ChEBI omits the title from the SD files and stores the id after the "> <ChEBI ID>" line. In that case, use `id_tag = "ChEBI ID"`.

The *reader_args* is a dictionary with additional structure reader parameters. The parameters depend on the toolkit and the format. Unknown parameters are ignored.

errors specifies how to handle errors. The value "strict" raises an exception if there are any detected errors. The value "report" sends an error message to stderr and skips to the next record. The value "ignore" skips to the next record.

Here is an example of using fingerprints generated from structure file:

```
from chemfp.bitops import hex_encode
fp_reader = chemfp.read_molecule_fingerprints("OpenBabel-FP4/1", "example.sdf.gz")
print("Each fingerprint has", fp_reader.metadata.num_bits, "bits")
for (id, fp) in fp_reader:
  print(id, hex_encode(fp))
```

See also *chemfp.read_molecule_fingerprints_from_string()*.

Parameters

- **type** (*string or Metadata*) – information about how to convert the input structure into a fingerprint
- **source** (*A filename (as a string), a file object, or None to read from stdin*) – The structure data source.
- **format** (*string, or None to autodetect based on the source*) – The file format and optional compression. Examples: "smi" and "sdf.gz"
- **id_tag** (*string, or None to use the default title for the given format*) – The tag containing the record id. Example: "ChEBI ID". Only valid for SD files.
- **reader_args** (*dict, or None to use the default arguments*) – additional parameters for the structure reader
- **errors** (*one of "strict", "report", or "ignore"*) – specify how to handle parse errors

Returns a *chemfp.FingerprintReader*

chemfp.**read_molecule_fingerprints_from_string**(*type*, *content*, *format*, *id_tag=None*, *reader_args=None*, *errors="strict"*)

Read structures from the content string and return the corresponding ids and fingerprints

The parameters are identical to *chemfp.read_molecule_fingerprints()* except that the entire content is passed through as a *content* string, rather than as a *source* filename. See that function for details.

You must specify the format! As there is no *source* filename, it's not possible to guess the format based on the extension, and there is no support for auto-detecting the format by looking at the string content.

Parameters

- **type** (*string or Metadata*) – information about how to convert the input structure into a fingerprint
- **content** (*string*) – The structure data as a string.
- **format** (*string*) – The file format and optional compression. Examples: "smi" and "sdf.gz"
- **id_tag** (*string, or None to use the default title for the given format*) – The tag containing the record id. Example: "ChEBI ID". Only valid for SD files.
- **reader_args** (*dict, or None to use the default arguments*) – additional parameters for the structure reader
- **errors** (*one of "strict" (raise exception), "report" (send a message to stderr and continue processing), or "ignore" (continue processing)*) – specify how to handle parse errors

Returns a *chemfp.FingerprintReader*

chemfp.**open_fingerprint_writer**(*destination*, *metadata=None*, *format=None*, *alignment=8*, *reorder=True*, *tmpdir=None*, *max_spool_size=None*, *errors="strict"*, *location=None*)

Create a fingerprint writer for the given destination

The fingerprint writer is an object with methods to write fingerprints to the given *destination*. The output format is based on the *format*. If that's None then the format depends on the *destination*, or is "fps" if the attempts at format detection fail.

The *metadata*, if given, is a *Metadata* instance, and used to fill the header of an FPS file or META block of an FPB file.

If the output format is "fps" or "fps.gz" then *destination* may be a filename, a file object, or None for stdout. If the output format is "fpb" then *destination* must be a filename.

Some options only apply to FPB output. The *alignment* specifies the arena byte alignment. By default the fingerprints are reordered by popcount, which enables sublinear similarity search. Set *reorder* to `False` to preserve the input fingerprint order.

The default FPB writer stores everything into memory before writing the file, which may cause performance problems if there isn't enough available free memory. In that case, set *max_spool_size* to the number of bytes of memory to use before spooling intermediate data to a file. (Note: there are two independent spools so this may use up to roughly twice as much memory as specified.)

Use *tmpdir* to specify where to write the temporary spool files if you don't want to use the operating system default. You may also set the TMPDIR, TEMP or TMP environment variables.

Some options only apply to FPS output. *errors* specifies how to handle recoverable write errors. The value "strict" raises an exception if there are any detected errors. The value "report" sends an error message to stderr and skips to the next record. The value "ignore" skips to the next record.

The *location* is a `Location` instance. It lets the caller access state information such as the number of records that have been written.

Parameters

- **destination** (*a filename, file object, or None*) – the output destination
- **metadata** (*a Metadata instance, or None*) – the fingerprint metadata
- **format** (*None, "fps", "fps.gz", or "fpb"*) – the output format
- **alignment** (*positive integer*) – arena byte alignment for FPB files
- **reorder** (*True or False*) – True reorders the fingerprints by popcount, False leaves them in input order
- **tmpdir** (*string or None*) – the directory to use for temporary files, when max_spool_size is specified
- **max_spool_size** (*integer, or None*) – number of bytes to store in memory before using a temporary file. If None, use memory for everything.
- **location** (*a Location instance, or None*) – a location object used to access output state information

Returns a *chemfp.FingerprintWriter*

ChemFPError

class chemfp.**ChemFPError**

Base class for all of the chemfp exceptions

ParseError

class chemfp.**ParseError**

Exception raised by the molecule and fingerprint parsers and writers

The public attributes are:

msg

a string describing the exception

location

a *chemfp.io.Location* instance, or None

Metadata

class chemfp.**Metadata**

Store information about a set of fingerprints

The public attributes are:

num_bits

the number of bits in the fingerprint

num_bytes

the number of bytes in the fingerprint

type
the fingerprint type string

aromaticity
aromaticity model (only used with OEChem, and now deprecated)

software
software used to make the fingerprints

sources
list of sources used to make the fingerprint

date
a datetime timestamp of when the fingerprints were made

__repr__()
Return a string like `Metadata(num_bits=1024, num_bytes=128, type='OpenBabel/FP2', ....)`

__str__()
Show the metadata in FPS header format

copy(*num_bits=None*, *num_bytes=None*, *type=None*, *aromaticity=None*, *software=None*, *sources=None*, *date=None*)
Return a new Metadata instance based on the current attributes and optional new values

When called with no parameter, make a new Metadata instance with the same attributes as the current instance.

If a given call parameter is not None then it will be used instead of the current value. If you want to change a current value to None then you will have to modify the new Metadata after you created it.

Parameters

- **num_bits** (*an integer, or None*) – the number of bits in the fingerprint
- **num_bytes** (*an integer, or None*) – the number of bytes in the fingerprint
- **type** (*string or None*) – the fingerprint type description
- **aromaticity** (*None*) – obsolete
- **software** (*string or None*) – a description of the software
- **sources** (*list of strings, a string (interpreted as a list with one string), or None*) – source filenames
- **date** (*a datetime instance, or None*) – creation or processing date for the contents

Returns a new Metadata instance

FingerprintReader

class chemfp.**FingerprintReader**

Base class for all chemfp objects holding fingerprint records

All FingerprintReader instances have a `metadata` attribute containing a Metadata and can be iteratated over to get the (id, fingerprint) for each record.

__iter__()
iterate over the (id, fingerprint) pairs

iter_arenas(*arena_size=1000*)
iterate through *arena_size* fingerprints at a time, as subarenas

Iterate through *arena_size* fingerprints at a time, returned as *chemfp.arena.FingerprintArena* instances. The arenas are in input order and not reordered by popcount.

This method helps trade off between performance and memory use. Working with arenas is often faster than processing one fingerprint at a time, but if the file is very large then you might run out of memory, or get bored while waiting to process all of the fingerprint before getting the first answer.

If *arena_size* is None then this makes an iterator which returns a single arena containing all of the fingerprints.

Parameters **arena_size** (*positive integer, or None*) – The number of fingerprints to put into each arena.

Returns an iterator of *chemfp.arena.FingerprintArena* instances

save(*destination*, *format=None*)
Save the fingerprints to a given destination and format

The output format is based on the *format*. If the format is None then the format depends on the *destination* file extension. If the extension isn't recognized then the fingerprints will be saved in "fps" format.

If the output format is "fps" or "fps.gz" then *destination* may be a filename, a file object, or None; None writes to stdout.

If the output format is "fpb" then *destination* must be a filename.

Parameters

- **destination** (*a filename, file object, or None*) – the output destination
- **format** (*None, "fps", "fps.gz", or "fpb"*) – the output format

Returns None

get_fingerprint_type()
Get the fingerprint type object based on the metadata's type field

This uses `self.metadata.type` to get the fingerprint type string then calls *chemfp.get_fingerprint_type()* to get and return a *chemfp.types.FingerprintType* instance.

This will raise a TypeError if there is no metadata, and a ValueError if the type field was invalid or the fingerprint type isn't available.

Returns a *chemfp.types.FingerprintType*

FingerprintIterator

class chemfp.**FingerprintIterator**

A *chemfp.FingerprintReader* for an iterator of (id, fingerprint) pairs

This is often used as an adapter container to hold the metadata and (id, fingerprint) iterator. It supports an optional location, and can call a close function when the iterator has completed.

A FingerprintIterator is a context manager which will close the underlying iterator if it's given a close handler.

Like all iterators you can use next() to get the next (id, fingerprint) pair.

__init__(*metadata*, *id_fp_iterator*, *location=None*, *close=None*)
Initialize with a Metadata instance and the (id, fingerprint) iterator

The *metadata* is a *Metadata* instance. The *id_fp_iterator* is an iterator which returns (id, fingerprint) pairs.

The optional *location* is a *chemfp.io.Location*. The optional *close* callable is called (as `close()`) whenever `self.close()` is called and when the context manager exits.

__iter__()
Iterate over the (id, fingerprint) pairs

close()
Close the iterator

The call will be forwarded to the `close` callable passed to the constructor. If that `close` is None then this does nothing.

Fingerprints

class chemfp.**Fingerprints**

A `chemf.FingerprintReader` containing a metadata and a list of (id, fingerprint) pairs.

This is typically used as an adapater when you have a list of (id, fingerprint) pairs and you want to pass it (and the metadata) to the rest of the chemfp API.

This implements a simple list-like collection of fingerprints. It supports:

- for (id, fingerprint) in fingerprints: ...
- id, fingerprint = fingerprints[1]
- len(fingerprints)

More features, like slicing, will be added as needed or when requested.

__init__(*metadata*, *id_fp_pairs*)
Initialize with a Metadata instance and the (id, fingerprint) pair list

The *metadata* is a *Metadata* instance. The *id_fp_iterator* is an iterator which returns (id, fingerprint) pairs.

FingerprintWriter

class chemfp.**FingerprintWriter**

Base class for the fingerprint writers

The three fingerprint writer classes are:

- *chemfp.fps_io.FPSWriter* - write an FPS file
- *chemfp.fpb_io.OrderedFPBWriter* - write an FPB file, sorted by popcount
- *chemfp.fpb_io.InputOrderFPBWriter* - write an FPB file, preserving input order

Use *chemfp.open_fingerprint_writer()* to create a fingerprint writer class; do not create them directly.

All classes have the following attributes:

- metadata - a *chemfp.Metadata* instance

•closed - False when the file is open, else True

Fingerprint writers are also their own context manager, and close the writer on context exit.

write_fingerprint(*id*, *fp*)
Write a single fingerprint record with the given id and fp to the destination

Parameters

- **id** (*string*) – the record identifier
- **fp** (*byte string*) – the fingerprint

write_fingerprints(*id_fp_pairs*)
Write a sequence of (id, fingerprint) pairs to the destination

Parameters **id_fp_pairs** – An iterable of (id, fingerprint) pairs. *id* is a string and *fingerprint* is a byte string.

close()
Close the writer

This will set self.closed to False.

ChemFPProblem

class chemfp.**ChemFPProblem**
Information about a compatibility problem between a query and target.

Instances are generated by *chemfp.check_fingerprint_problems()* and *chemfp.check_metadata_problems()*.

The public attributes are:

severity
one of "info", "warning", or "error"

error_level
5 for "info", 10 for "warning", and 20 for "error"

category
a string used as a category name. This string will not change over time.

description
a more detailed description of the error, including details of the mismatch. The description depends on *query_name* and *target_name* and may change over time.

The current category names are:

- "num_bits mismatch" (error)
- "num_bytes_mismatch" (error)
- "type mismatch" (warning)
- "aromaticity mismatch" (info)
- "software mismatch" (info)

chemfp.**check_fingerprint_problems**(*query_fp*, *target_metadata*, *query_name="query"*, *target_name="target"*)
Return a list of compatibility problems between a fingerprint and a metadata

If there are no problems then this returns an empty list. If there is a bit length or byte length mismatch between the *query_fp* byte string and the *target_metadata* then it will return a list containing a *ChemFPProblem* instance, with a severity level "error" and category "num_bytes mismatch".

This function is usually used to check if a query fingerprint is compatible with the target fingerprints. In case of a problem, the default message looks like:

```
>>> problems = check_fingerprint_problems("A"*64, Metadata(num_bytes=128))
>>> problems[0].description
'query contains 64 bytes but target has 128 byte fingerprints'
```

You can change the error message with the *query_name* and *target_name* parameters:

```
>>> import chemfp
>>> problems = check_fingerprint_problems("z"*64, chemfp.Metadata(num_bytes=128),
...       query_name="input", target_name="database")
>>> problems[0].description
'input contains 64 bytes but database has 128 byte fingerprints'
```

Parameters

- **query_fp** (*byte string*) – a fingerprint (usually the query fingerprint)
- **target_metadata** (*Metadata instance*) – the metadata to check against (usually the target metadata)
- **query_name** (*string*) – the text used to describe the fingerprint, in case of problem
- **target_name** (*string*) – the text used to describe the metadata, in case of problem

Returns a list of *ChemFPProblem* instances

chemfp.**check_metadata_problems**(*query_metadata*, *target_metadata*, *query_name="query"*, *target_name="target"*)

Return a list of compatibility problems between two metadata instances.

If there are no probelms then this returns an empty list. Otherwise it returns a list of *ChemFPProblem* instances, with a severity level ranging from "info" to "error".

Bit length and byte length mismatches produce an "error". Fingerprint type and aromaticity mismatches produce a "warning". Software version mismatches produce an "info".

This is usually used to check if the query metadata is incompatible with the target metadata. In case of a problem the messages look like:

```
>>> import chemfp
>>> m1 = chemfp.Metadata(num_bytes=128, type="Example/1")
>>> m2 = chemfp.Metadata(num_bytes=256, type="Counter-Example/1")
>>> problems = chemfp.check_metadata_problems(m1, m2)
>>> len(problems)
2
>>> print(problems[1].description)
query has fingerprints of type 'Example/1' but target has fingerprints of type
↪'Counter-Example/1'
```

You can change the error message with the *query_name* and *target_name* parameters:

```
>>> problems = chemfp.check_metadata_problems(m1, m2, query_name="input", target_
↪name="database")
>>> print(problems[1].description)
input has fingerprints of type 'Example/1' but database has fingerprints of type
↪'Counter-Example/1'
```

Parameters

- **fp** (*byte string*) – a fingerprint
- **metadata** (*Metadata instance*) – the metadata to check against
- **query_name** (*string*) – the text used to describe the fingerprint, in case of problem
- **target_name** (*string*) – the text used to describe the metadata, in case of problem

Returns a list of *ChemFPProblem* instances

chemfp.**count_tanimoto_hits**(*queries*, *targets*, *threshold=0.7*, *arena_size=100*)

Count the number of targets within *threshold* of each query term

For each query in *queries*, count the number of targets in *targets* which are at least *threshold* similar to the query. This function returns an iterator containing the (query_id, count) pairs.

Example:

```
queries = chemfp.open("queries.fps")
targets = chemfp.load_fingerprints("targets.fps.gz")
for (query_id, count) in chemfp.count_tanimoto_hits(queries, targets, threshold=0.
↪9):
  print(query_id, "has", count, "neighbors with at least 0.9 similarity")
```

Internally, queries are processed in batches with *arena_size* elements. A small batch size uses less overall memory and has lower processing latency, while a large batch size has better overall performance. Use arena_size=None to process the input as a single batch.

Note: an *chemfp.fps_io.FPSReader* may be used as a target but it will only process one batch and not reset for the next batch. It's faster to search a *chemfp.arena.FingerprintArena*, but if you have an FPS file then that takes extra time to load. At times, if there is a small number of queries, the time to load the arena from an FPS file may be slower than the direct search using an FPSReader.

If you know the targets are in an arena then you may want to use *chemfp.search.count_tanimoto_hits_fp()* or *chemfp.search.count_tanimoto_hits_arena()*.

Parameters

- **queries** (*any fingerprint container*) – The query fingerprints.
- **targets** (*chemfp.arena.FingerprintArena* or the slower *chemfp.fps_io.FPSReader*) – The target fingerprints.
- **threshold** (*float between 0.0 and 1.0, inclusive*) – The minimum score threshold.
- **arena_size** (*a positive integer, or None*) – The number of queries to process in a batch

Returns iterator of the (query_id, score) pairs, one for each query

chemfp.**count_tanimoto_hits_symmetric**(*fingerprints*, *threshold=0.7*)

Find the number of other fingerprints within *threshold* of each fingerprint

For each fingerprint in the *fingerprints* arena, find the number of other fingerprints in the same arena which are at least *threshold* similar to it. The arena must have pre-computed popcounts. A fingerprint never matches itself.

This function returns an iterator of (fingerprint_id, count) pairs.

Example:

```
arena = chemfp.load_fingerprints("targets.fps.gz")
for (fp_id, count) in chemfp.count_tanimoto_hits_symmetric(arena, threshold=0.6):
    print(fp_id, "has", count, "neighbors with at least 0.6 similarity")
```

You may also be interested in *chemfp.search.count_tanimoto_hits_symmetric()*.

Parameters

- **fingerprints** (*a FingerprintArena with precomputed popcount_indices*) – The arena containing the fingerprints.
- **threshold** (*float between 0.0 and 1.0, inclusive*) – The minimum score threshold.

Returns An iterator of (fp_id, count) pairs, one for each fingerprint

chemfp.**threshold_tanimoto_search**(*queries*, *targets*, *threshold=0.7*, *arena_size=100*)

Find all targets within *threshold* of each query term

For each query in *queries*, find all the targets in *targets* which are at least *threshold* similar to the query. This function returns an iterator containing the (query_id, hits) pairs. The hits are stored as a list of (target_id, score) pairs.

Example:

```
queries = chemfp.open("queries.fps")
targets = chemfp.load_fingerprints("targets.fps.gz")
for (query_id, hits) in chemfp.id_threshold_tanimoto_search(queries, targets,␣
↪threshold=0.8):
    print(query_id, "has", len(hits), "neighbors with at least 0.8 similarity")
    non_identical = [target_id for (target_id, score) in hits if score != 1.0]
    print("  The non-identical hits are:", non_identical)
```

Internally, queries are processed in batches with *arena_size* elements. A small batch size uses less overall memory and has lower processing latency, while a large batch size has better overall performance. Use `arena_size=None` to process the input as a single batch.

Note: an *chemfp.fps_io.FPSReader* may be used as a target but it will only process one batch and not reset for the next batch. It's faster to search a *chemfp.arena.FingerprintArena*, but if you have an FPS file then that takes extra time to load. At times, if there is a small number of queries, the time to load the arena from an FPS file may be slower than the direct search using an FPSReader.

If you know the targets are in an arena then you may want to use *chemfp.search.threshold_tanimoto_search_fp()* or *chemfp.search.threshold_tanimoto_search_arena()*.

Parameters

- **queries** (*any fingerprint container*) – The query fingerprints.
- **targets** (*chemfp.arena.FingerprintArena* or the slower *chemfp.fps_io.FPSReader*) – The target fingerprints.
- **threshold** (*float between 0.0 and 1.0, inclusive*) – The minimum score threshold.
- **arena_size** (*positive integer, or None*) – The number of queries to process in a batch

Returns An iterator containing (query_id, hits) pairs, one for each query. 'hits' contains a list of (target_id, score) pairs.

chemfp.**threshold_tanimoto_search_symmetric**(*fingerprints*, *threshold=0.7*)

Find the other fingerprints within *threshold* of each fingerprint

For each fingerprint in the *fingerprints* arena, find the other fingerprints in the same arena which share at least *threshold* similar to it. The arena must have pre-computed popcounts. A fingerprint never matches itself.

This function returns an iterator of (fingerprint, SearchResult) pairs. The *chemfp.search.SearchResult* hit order is arbitrary.

Example:

```
arena = chemfp.load_fingerprints("targets.fps.gz")
for (fp_id, hits) in chemfp.threshold_tanimoto_search_symmetric(arena,␣
↪threshold=0.75):
    print(fp_id, "has", len(hits), "neighbors:")
    for (other_id, score) in hits.get_ids_and_scores():
        print("   %s  %.2f" % (other_id, score))
```

You may also be interested in the *chemfp.search.threshold_tanimoto_search_symmetric()* function.

Parameters

- **fingerprints** (*a FingerprintArena with precomputed popcount_indices*) – The arena containing the fingerprints.
- **threshold** (*float between 0.0 and 1.0, inclusive*) – The minimum score threshold.

Returns An iterator of (fp_id, SearchResult) pairs, one for each fingerprint

chemfp.**knearest_tanimoto_search**(*queries*, *targets*, *k=3*, *threshold=0.7*, *arena_size=100*)

Find the *k*-nearest targets within *threshold* of each query term

For each query in *queries*, find the *k*-nearest of all the targets in *targets* which are at least *threshold* similar to the query. Ties are broken arbitrarily and hits with scores equal to the smallest value may have been omitted.

This function returns an iterator containing the (query_id, hits) pairs, where hits is a list of (target_id, score) pairs, sorted so that the highest scores are first. The order of ties is arbitrary.

Example:

```
# Use the first 5 fingerprints as the queries
queries = next(chemfp.open("pubchem_subset.fps").iter_arenas(5))
targets = chemfp.load_fingerprints("pubchem_subset.fps")

# Find the 3 nearest hits with a similarity of at least 0.8
for (query_id, hits) in chemfp.id_knearest_tanimoto_search(queries, targets, k=3,␣
↪threshold=0.8):
    print(query_id, "has", len(hits), "neighbors with at least 0.8 similarity")
    if hits:
        target_id, score = hits[-1]
        print("    The least similar is", target_id, "with score", score)
```

Internally, queries are processed in batches with *arena_size* elements. A small batch size uses less overall memory and has lower processing latency, while a large batch size has better overall performance. Use `arena_size=None` to process the input as a single batch.

Note: an *chemfp.fps_io.FPSReader* may be used as a target but it will only process one batch and not reset for the next batch. It's faster to search a *chemfp.arena.FingerprintArena*, but if you have an FPS file then that takes extra time to load. At times, if there is a small number of queries, the time to load the arena from an FPS file may be slower than the direct search using an FPSReader.

If you know the targets are in an arena then you may want to use *chemfp.search.knearest_tanimoto_search_fp()* or *chemfp.search.knearest_tanimoto_search_arena()*.

Parameters

- **queries** (*any fingerprint container*) – The query fingerprints.
- **targets** (*chemfp.arena.FingerprintArena* or the slower *chemfp.fps_io.FPSReader*) – The target fingerprints.
- **k** (*positive integer*) – The maximum number of nearest neighbors to find.
- **threshold** (*float between 0.0 and 1.0, inclusive*) – The minimum score threshold.
- **arena_size** (*positive integer, or None*) – The number of queries to process in a batch

Returns An iterator containing (query_id, hits) pairs, one for each query. The *hits* are a list of (target_id, score) pairs, sorted by score.

chemfp.**knearest_tanimoto_search_symmetric**(*fingerprints*, *k=3*, *threshold=0.7*)

Find the *k*-nearest fingerprints within *threshold* of each fingerprint

For each fingerprint in the *fingerprints* arena, find the nearest *k* fingerprints in the same arena which have at least *threshold* similar to it. The arena must have pre-computed popcounts. A fingerprint never matches itself.

This function returns an iterator of (fingerprint, SearchResult) pairs. The *chemfp.search.SearchResult* hits are ordered from highest score to lowest, with ties broken arbitrarily.

Example:

```
arena = chemfp.load_fingerprints("targets.fps.gz")
for (fp_id, hits) in chemfp.knearest_tanimoto_search_symmetric(arena, k=5,␣
↪threshold=0.5):
    print(fp_id, "has", len(hits), "neighbors, with scores", end="")
    print(", ".join("%.2f" % x for x in hits.get_scores()))
```

You may also be interested in the *chemfp.search.knearest_tanimoto_search_symmetric()* function.

Parameters

- **fingerprints** (*a FingerprintArena with precomputed popcount_indices*) – The arena containing the fingerprints.
- **k** (*positive integer*) – The maximum number of nearest neighbors to find.
- **threshold** (*float between 0.0 and 1.0, inclusive*) – The minimum score threshold.

Returns An iterator of (fp_id, SearchResult) pairs, one for each fingerprint

chemfp.**count_tversky_hits**(*queries*, *targets*, *threshold=0.7*, *alpha=1.0*, *beta=1.0*, *arena_size=100*)

Count the number of targets within *threshold* of each query term

For each query in *queries*, count the number of targets in *targets* which are at least *threshold* similar to the query. This function returns an iterator containing the (query_id, count) pairs.

Example:

```
queries = chemfp.open("queries.fps")
targets = chemfp.load_fingerprints("targets.fps.gz")
for (query_id, count) in chemfp.count_tversky_hits(
          queries, targets, threshold=0.9, alpha=0.5, beta=0.5):
  print(query_id, "has", count, "neighbors with at least 0.9 Dice similarity")
```

Internally, queries are processed in batches with *arena_size* elements. A small batch size uses less overall memory and has lower processing latency, while a large batch size has better overall performance. Use arena_size=None to process the input as a single batch.

Note: an *chemfp.fps_io.FPSReader* may be used as a target but it will only process one batch and not reset for the next batch. It's faster to search a *chemfp.arena.FingerprintArena*, but if you have an FPS file then that takes extra time to load. At times, if there is a small number of queries, the time to load the arena from an FPS file may be slower than the direct search using an FPSReader.

If you know the targets are in an arena then you may want to use *chemfp.search.count_tversky_hits_fp()* or *chemfp.search.count_tversky_hits_arena()*.

Parameters

- **queries** (*any fingerprint container*) – The query fingerprints.
- **targets** (*chemfp.arena.FingerprintArena* or the slower *chemfp.fps_io.FPSReader*) – The target fingerprints.
- **threshold** (*float between 0.0 and 1.0, inclusive*) – The minimum score threshold.
- **arena_size** (*a positive integer, or None*) – The number of queries to process in a batch

Returns iterator of the (query_id, score) pairs, one for each query

chemfp.**count_tversky_hits_symmetric**(*fingerprints*, *threshold=0.7*, *alpha=1.0*, *beta=1.0*)

Find the number of other fingerprints within *threshold* of each fingerprint

For each fingerprint in the *fingerprints* arena, find the number of other fingerprints in the same arena which are at least *threshold* similar to it. The arena must have pre-computed popcounts. A fingerprint never matches itself.

This function returns an iterator of (fingerprint_id, count) pairs.

Example:

```
arena = chemfp.load_fingerprints("targets.fps.gz")
for (fp_id, count) in chemfp.count_tversky_hits_symmetric(
        arena, threshold=0.6, alpha=0.5, beta=0.5):
    print(fp_id, "has", count, "neighbors with at least 0.6 Dice similarity")
```

You may also be interested in *chemfp.search.count_tversky_hits_symmetric()*.

Parameters

- **fingerprints** (*a FingerprintArena with precomputed popcount_indices*) – The arena containing the fingerprints.
- **threshold** (*float between 0.0 and 1.0, inclusive*) – The minimum score threshold.

Returns An iterator of (fp_id, count) pairs, one for each fingerprint

chemfp.**threshold_tversky_search**(*queries*, *targets*, *threshold=0.7*, *alpha=1.0*, *beta=1.0*, *arena_size=100*)

Find all targets within *threshold* of each query term

For each query in *queries*, find all the targets in *targets* which are at least *threshold* similar to the query. This function returns an iterator containing the (query_id, hits) pairs. The hits are stored as a list of (target_id, score) pairs.

Example:

```
queries = chemfp.open("queries.fps")
targets = chemfp.load_fingerprints("targets.fps.gz")
for (query_id, hits) in chemfp.id_threshold_tanimoto_search(
            queries, targets, threshold=0.8, alpha=0.5, beta=0.5):
    print(query_id, "has", len(hits), "neighbors with at least 0.8 Dice similarity
↪")
    non_identical = [target_id for (target_id, score) in hits if score != 1.0]
    print("  The non-identical hits are:", non_identical)
```

Internally, queries are processed in batches with *arena_size* elements. A small batch size uses less overall memory and has lower processing latency, while a large batch size has better overall performance. Use `arena_size=None` to process the input as a single batch.

Note: an *chemfp.fps_io.FPSReader* may be used as a target but it will only process one batch and not reset for the next batch. It's faster to search a *chemfp.arena.FingerprintArena*, but if you have an FPS file then that takes extra time to load. At times, if there is a small number of queries, the time to load the arena from an FPS file may be slower than the direct search using an FPSReader.

If you know the targets are in an arena then you may want to use *chemfp.search.threshold_tversky_search_fp()* or *chemfp.search.threshold_tversky_search_arena()*.

Parameters

- **queries** (*any fingerprint container*) – The query fingerprints.
- **targets** (*chemfp.arena.FingerprintArena* or the slower *chemfp.fps_io.FPSReader*) – The target fingerprints.
- **threshold** (*float between 0.0 and 1.0, inclusive*) – The minimum score threshold.
- **arena_size** (*positive integer, or None*) – The number of queries to process in a batch

Returns An iterator containing (query_id, hits) pairs, one for each query. 'hits' contains a list of (target_id, score) pairs.

chemfp.**threshold_tversky_search_symmetric**(*fingerprints*, *threshold=0.7*, *alpha=1.0*, *beta=1.0*)

Find the other fingerprints within *threshold* of each fingerprint

For each fingerprint in the *fingerprints* arena, find the other fingerprints in the same arena which share at least *threshold* similar to it. The arena must have pre-computed popcounts. A fingerprint never matches itself.

This function returns an iterator of (fingerprint, SearchResult) pairs. The *chemfp.search.SearchResult* hit order is arbitrary.

Example:

```
arena = chemfp.load_fingerprints("targets.fps.gz")
for (fp_id, hits) in chemfp.threshold_tversky_search_symmetric(
            arena, threshold=0.75, alpha=0.5, beta=0.5):
    print(fp_id, "has", len(hits), "Dice neighbors:")
    for (other_id, score) in hits.get_ids_and_scores():
        print("   %s  %.2f" % (other_id, score))
```

You may also be interested in the *chemfp.search.threshold_tversky_search_symmetric()* function.

Parameters

- **fingerprints** (*a FingerprintArena with precomputed popcount_indices*) – The arena containing the fingerprints.
- **threshold** (*float between 0.0 and 1.0, inclusive*) – The minimum score threshold.

Returns An iterator of (fp_id, SearchResult) pairs, one for each fingerprint

chemfp.**knearest_tversky_search**(*queries*, *targets*, *k=3*, *threshold=0.7*, *alpha=1.0*, *beta=1.0*, *arena_size=100*)

Find the *k*-nearest targets within *threshold* of each query term

For each query in *queries*, find the *k*-nearest of all the targets in *targets* which are at least *threshold* similar to the query. Ties are broken arbitrarily and hits with scores equal to the smallest value may have been omitted.

This function returns an iterator containing the (query_id, hits) pairs, where hits is a list of (target_id, score) pairs, sorted so that the highest scores are first. The order of ties is arbitrary.

Example:

```
# Use the first 5 fingerprints as the queries
queries = next(chemfp.open("pubchem_subset.fps").iter_arenas(5))
targets = chemfp.load_fingerprints("pubchem_subset.fps")

# Find the 3 nearest hits with a similarity of at least 0.8
for (query_id, hits) in chemfp.id_knearest_tversky_search(
          queries, targets, k=3, threshold=0.8, alpha=0.5, beta=0.5):
    print(query_id, "has", len(hits), "neighbors with at least 0.8 Dice similarity
↪")
    if hits:
        target_id, score = hits[-1]
        print("    The least similar is", target_id, "with score", score)
```

Internally, queries are processed in batches with *arena_size* elements. A small batch size uses less overall memory and has lower processing latency, while a large batch size has better overall performance. Use `arena_size=None` to process the input as a single batch.

Note: an *chemfp.fps_io.FPSReader* may be used as a target but it will only process one batch and not reset for the next batch. It's faster to search a *chemfp.arena.FingerprintArena*, but if you have an FPS file then that takes extra time to load. At times, if there is a small number of queries, the time to load the arena from an FPS file may be slower than the direct search using an FPSReader.

If you know the targets are in an arena then you may want to use *chemfp.search.knearest_tversky_search_fp()* or *chemfp.search.knearest_tversky_search_arena()*.

Parameters

- **queries** (*any fingerprint container*) – The query fingerprints.
- **targets** (*chemfp.arena.FingerprintArena* or the slower *chemfp.fps_io.FPSReader*) – The target fingerprints.
- **k** (*positive integer*) – The maximum number of nearest neighbors to find.
- **threshold** (*float between 0.0 and 1.0, inclusive*) – The minimum score threshold.

- **arena_size** (*positive integer, or None*) – The number of queries to process in a batch

Returns An iterator containing (query_id, hits) pairs, one for each query. The *hits* are a list of (target_id, score) pairs, sorted by score.

chemfp.**knearest_tversky_search_symmetric**(*fingerprints*, *k=3*, *threshold=0.7*, *alpha=1.0*, *beta=1.0*)

Find the *k*-nearest fingerprints within *threshold* of each fingerprint

For each fingerprint in the *fingerprints* arena, find the nearest *k* fingerprints in the same arena which have at least *threshold* similar to it. The arena must have pre-computed popcounts. A fingerprint never matches itself.

This function returns an iterator of (fingerprint, SearchResult) pairs. The *chemfp.search.SearchResult* hits are ordered from highest score to lowest, with ties broken arbitrarily.

Example:

```
arena = chemfp.load_fingerprints("targets.fps.gz")
for (fp_id, hits) in chemfp.knearest_tversky_search_symmetric(
        arena, k=5, threshold=0.5, alpha=0.5, beta=0.5):
    print(fp_id, "has", len(hits), "neighbors, with Dice scores", end="")
    print(", ".join("%.2f" % x for x in hits.get_scores()))
```

You may also be interested in the *chemfp.search.knearest_tversky_search_symmetric()* function.

Parameters

- **fingerprints** (*a FingerprintArena with precomputed popcount_indices*) – The arena containing the fingerprints.
- **k** (*positive integer*) – The maximum number of nearest neighbors to find.
- **threshold** (*float between 0.0 and 1.0, inclusive*) – The minimum score threshold.

Returns An iterator of (fp_id, SearchResult) pairs, one for each fingerprint

chemfp.**get_fingerprint_families**()

Return a list of available fingerprint families

Returns a list of *chemfp.types.FingerprintFamily* instances

chemfp.**get_fingerprint_family**(*family_name*)

Return the named fingerprint family, or raise a ValueError if not available

Given a *family_name* like `OpenBabel-FP2` or `OpenEye-MACCS166` return the corresponding *chemfp.types.FingerprintFamily*.

Parameters **family_name** (*string*) – the family name

Returns a *chemfp.types.FingerprintFamily* instance

chemfp.**get_fingerprint_family_names**(*include_unavailable=False*)

Return a set of fingerprint family name strings

The function tries to load each known fingerprint family. The names of the families which could be loaded are returned as a set of strings.

If *include_unavailable* is True then this will return a set of all of the fingerprint family names, including those which could not be loaded.

The set contains both the versioned and unversioned family names, so both `OpenBabel-FP2/1` and `OpenBabel-FP2` may be returned.

Parameters **include_unavailable** (*True or False*) – Should unavailable family names be included in the result set?

Returns a set of strings

chemfp.**get_fingerprint_type**(*type*, *fingerprint_kwargs=None*)

Get the fingerprint type based on its type string and optional keyword arguments

Given a fingerprint *type* string like `OpenBabel-FP2`, or `RDKit-Fingerprint/1 fpSize=1024`, return the corresponding *chemfp.types.FingerprintType*.

The fingerprint type string may include fingerprint parameters. Parameters can also be specified through the *fingerprint_kwargs* dictionary, where the dictionary values are native Python values. If the same parameter is specified in the type string and the kwargs dictionary then the *fingerprint_kwargs* takes precedence.

For example:

```
>>> fptype = get_fingerprint_type("RDKit-Fingerprint fpSize=1024 minPath=3", {
↪"fpSize": 4096})
>>> fptype.get_type()
'RDKit-Fingerprint/2 minPath=3 maxPath=7 fpSize=4096 nBitsPerHash=2 useHs=1'
```

Use *get_fingerprint_type_from_text_settings()* if your fingerprint parameter values are all string-encoded, eg, from the command-line or a configuration file.

Parameters

- **type** (*string*) – a fingerprint type string
- **fingerprint_kwargs** (*a dictionary of string names and Python types for values*) – fingerprint type parameters

Returns a *chemfp.types.FingerprintType*

chemfp.**get_fingerprint_type_from_text_settings**(*type*, *settings=None*)

Get the fingerprint type based on its type string and optional settings arguments

Given a fingerprint *type* string like `OpenBabel-FP2`, or `RDKit-Fingerprint/1 fpSize=1024`, return the corresponding *chemfp.types.FingerprintType*.

The fingerprint type string may include fingerprint parameters. Parameters can also be specified through the *settings* dictionary, where the dictionary values are string-encoded values. If the same parameter is specified in the *type* string and the *settings* dictionary then the *settings* take precedence.

For example:

```
>>> fptype = get_fingerprint_type_from_text_settings("RDKit-Fingerprint␣
↪fpSize=1024 minPath=3",
...                                                 {"fpSize": "4096"})
>>> fptype.get_type()
'RDKit-Fingerprint/2 minPath=3 maxPath=7 fpSize=4096 nBitsPerHash=2 useHs=1'
```

This function is for string settings from a configuration file or command-line. Use *get_fingerprint_type()* if your fingerprint parameters are Python values.

Parameters

- **type** (*string*) – a fingerprint type string
- **fingerprint_kwargs** (*a dictionary of string names and Python types for values*) – fingerprint type parameters

Returns a *chemfp.types.FingerprintType*

chemfp.**has_fingerprint_family**(*family_name*)
Test if the fingerprint family is available

Return True if the fingerprint *family_name* is available, otherwise False. The *family_name* may be versioned or unversioned, like "OpenBabel-FP2/1" or "OpenEye-MACCS166".

Parameters **family_name** (*string*) – the family name

Returns True or False

chemfp.**get_max_threads**()
Return the maximum number of threads available.

WARNING: this likely doesn't do what you think it does. Do not use!

If OpenMP is not available then this will return 1. Otherwise it returns the maximum number of threads available, as reported by omp_get_num_threads().

chemfp.**get_num_threads**()
Return the number of OpenMP threads to use in searches

Initially this is the value returned by omp_get_max_threads(), which is generally 4 unless you set the environment variable OMP_NUM_THREADS to some other value.

It may be any value in the range 1 to get_max_threads(), inclusive.

Returns the current number of OpenMP threads to use

chemfp.**set_num_threads**(*num_threads*)
Set the number of OpenMP threads to use in searches

If *num_threads* is less than one then it is treated as one, and a value greater than get_max_threads() is treated as get_max_threads().

Parameters **num_threads** (*int*) – the new number of OpenMP threads to use

chemfp.**get_toolkit**(*toolkit_name*)
Return the named toolkit, if available, or raise a ValueError

If *toolkit_name* is one of "openbabel", "openeye", or "rdkit" and the named toolkit is available, then it will return *chemfp.openbabel_toolkit*, *chemfp.openeye_toolkit*, or *chemfp.rdkit_toolkit*, respectively.:

```
>>> import chemfp
>>> chemfp.get_toolkit("openeye")
<module 'chemfp.openeye_toolkit' from 'chemfp/openeye_toolkit.py'>
>>> chemfp.get_toolkit("rdkit")
Traceback (most recent call last):
     ...
ValueError: Unable to get toolkit 'rdkit': No module named rdkit
```

Parameters **toolkit_name** (*string*) – the toolkit name

Returns the chemfp toolkit

Raises ValueError if *toolkit_name* is unknown or the toolkit does not exist

chemfp.**get_toolkit_names**()
Return a set of available toolkit names

The function checks if each supported toolkit is available by trying to import its corresponding module. It returns a set of toolkit names:

```
>>> import chemfp
>>> chemfp.get_toolkit_names()
set(['openeye', 'rdkit', 'openbabel'])
```

Returns a set of toolkit names, as strings

chemfp.**has_toolkit**(*toolkit_name*)

Return True if the named toolkit is available, otherwise False

If *toolkit_name* is one of "openbabel", "openeye", or "rdkit" then this function will test to see if the given toolkit is available, and if so return True. Otherwise it returns False.

```
>>> import chemfp
>>> chemfp.has_toolkit("openeye")
True
>>> chemfp.has_toolkit("openbabel")
False
```

The initial test for a toolkit can be slow, especially if the underlying toolkit loads a lot of shared libraries. The test is only done once, and cached.

Parameters **toolkit_name** (*string*) – the toolkit name

Returns True or False

chemfp.types - fingerprint families and types

A "fingerprint type" is an object which knows how to convert a molecule into a fingerprint. A "fingerprint family" is an object which uses a set of parameters to make a specific fingerprint type.

```
>>> import chemfp
>>> fpfamily = chemfp.get_fingerprint_family("RDKit-Fingerprint")
>>> fpfamily.get_defaults()
{'maxPath': 7, 'fpSize': 2048, 'nBitsPerHash': 2, 'minPath': 1, 'useHs': 1}
>>>
>>> fptype = fpfamily()  # create the default fingerprint type
>>> fptype.get_type()
'RDKit-Fingerprint/2 minPath=1 maxPath=7 fpSize=2048 nBitsPerHash=2 useHs=1'
>>>
>>> fptype = fpfamily(fpSize=1024)   # use a non-default value
>>> fptype.get_type()
'RDKit-Fingerprint/2  minPath=1 maxPath=7 fpSize=1024 nBitsPerHash=2 useHs=1'
>>> mol = fptype.toolkit.parse_molecule("c1ccccc1O", "smistring")
>>> fptype.compute_fingerprint(mol)
'\x04\x00\x00\x00\x00\x00\x10\x00\x00\x00  ... x00\x00\x00\x00\x00'
```

Fingerprint family class

FingerprintFamily

class chemfp.types.**FingerprintFamily**

A FingerprintFamily is used to create a FingerprintType or get information about its parameters

Two reasons to use a FingerprintFamily (instead of using *chemfp.get_fingerprint_type()* or *chemfp.get_fingerprint_type_from_text_settings()*) are:

- figure out the default arguments;
- given a text settings or parameter dictionary, use the keys from the default argument keys to remove other parameters before creating a FingerprintType (otherwise the creation function will raise an exception)

All fingerprint families have the following attributes:

- name - the type name, including version
- toolkit - the toolkit API for the underlying chemistry toolkit, or None

__repr__()
Return a string like 'FingerprintFamily(<RDKit-Fingerprint/2>)'

name
Read-only attribute.

The full fingerprint name, including the version

base_name
Read-only attribute.

The base fingerprint name, without the version

version
Read-only attribute.

The fingerprint version

toolkit
Read-only attribute.

The toolkit used to implement this fingerprint, or None

__call__(***fingerprint_kwargs*)
Create a fingerprint type; keyword arguments can override the defaults

The argument values are native Python values, not string-encoded values:

```
>>> import chemfp
>>> family = chemfp.get_fingerprint_family("RDKit-Fingerprint")
>>> fptype = family()
>>> fptype.get_type()
'RDKit-Fingerprint/2 minPath=1 maxPath=7 fpSize=2048 nBitsPerHash=2 useHs=1'
>>> fptype = family(fpSize=1024)
>>> fptype.get_type()
'RDKit-Fingerprint/2 minPath=1 maxPath=7 fpSize=1024 nBitsPerHash=2 useHs=1'
```

The function will raise an exception for unknown arguments.

Parameters **fingerprint_kwargs** – the fingerprint parameters

Returns an object implementing the *chemfp.types.FingerprintType* API

from_kwargs(*fingerprint_kwargs=None*)
Create a fingerprint type; items in the *fingerprint_kwargs* dictionary can override the defaults

The dictionary values are native Python values, not string-encoded values:

```
>>> import chemfp
>>> family = chemfp.get_fingerprint_family("RDKit-Fingerprint")
>>> fptype = family()
>>> fptype.get_type()
'RDKit-Fingerprint/2 minPath=1 maxPath=7 fpSize=2048 nBitsPerHash=2 useHs=1'
>>> fptype = family.from_kwargs({"fpSize": 1024})
>>> fptype.get_type()
'RDKit-Fingerprint/2 minPath=1 maxPath=7 fpSize=1024 nBitsPerHash=2 useHs=1'
```

The function will raise an exception for unknown arguments.

Parameters **fingerprint_kwargs** (*a dictionary where the values are Python objects*) – the fingerprint parameters

Returns an object implementing the *chemfp.types.FingerprintType* API

from_text_settings(*settings=None*)

Create a fingerprint type; *settings* is a dictionary with string-encoded value that can override the defaults

The dictionary values are string-encoded values, not native Python values. This function exists to help handle command-line arguments and setting files.:

```
>>> import chemfp
>>> family = chemfp.get_fingerprint_family("RDKit-Fingerprint")
>>> fptype = family.from_text_settings()
>>> fptype.get_type()
'RDKit-Fingerprint/2 minPath=1 maxPath=7 fpSize=2048 nBitsPerHash=2 useHs=1'
>>> fptype = family.from_text_settings({"fpSize": "1024"})
>>> fptype.get_type()
'RDKit-Fingerprint/2 minPath=1 maxPath=7 fpSize=1024 nBitsPerHash=2 useHs=1'
```

The function will raise an exception for unknown arguments.

Parameters **settings** (*a dictionary where the values are string-encoded*) – the fingerprint text settings

Returns an object implementing the *chemfp.types.FingerprintType* API

get_kwargs_from_text_settings(*settings=None*)

Convert a dictionary of string-encoded fingerprint parameters into native Python values

String-encoded values ("text settings") can come from the command-line, a configuration file, a web reqest, or other text sources. The fingerprint types need actual Python values. This method converts the first to the second:

```
>>> import chemfp
>>> family = chemfp.get_fingerprint_family("RDKit-Fingerprint")
>>> family.get_kwargs_from_text_settings()
{'maxPath': 7, 'fpSize': 2048, 'nBitsPerHash': 2, 'minPath': 1, 'useHs': 1}
>>> family.get_kwargs_from_text_settings({"fpSize": "128", "maxPath": "5"})
{'maxPath': 5, 'fpSize': 128, 'nBitsPerHash': 2, 'minPath': 1, 'useHs': 1}
```

Parameters **settings** (*a dictionary where the values are string-encoded*) – the fingerprint text settings

Returns an dictionary of (decoded) fingerprint parameters

get_defaults()

Return the default parameters as a dictionary

The dictionary values are native Python objects:

```
>>> import chemfp
>>> family = chemfp.get_fingerprint_family("RDKit-Fingerprint")
>>> family.get_defaults()
{'maxPath': 7, 'fpSize': 2048, 'nBitsPerHash': 2, 'minPath': 1, 'useHs': 1}
```

Returns an dictionary of fingerprint parameters

Base fingerprint type

FingerprintType

class chemfp.types.**FingerprintType**

The base to all fingerprint types

A fingerprint type has the following public attributes:

name
the fingerprint name, including the version

base_name
the fingerprint name, without the version

version
the fingerprint version

toolkit
the toolkit API for the underlying chemistry toolkit, or None

software
a string which characterizes the toolkit, including version information

num_bits
the number of bits in this fingerprint type

fingerprint_kwargs
a dictionary of the fingerprint arguments

The built-in fingerprint types are:

- *chemfp.openbabel_types.OpenBabelFP2FingerprintType_v1* - OpenBabel-FP2/1 - Open Babel FP2
- *chemfp.openbabel_types.OpenBabelFP3FingerprintType_v1* - OpenBabel-FP3/1 - Open Babel FP3
- *chemfp.openbabel_types.OpenBabelFP4FingerprintType_v1* - OpenBabel-FP4/1 - Open Babel FP4
- *chemfp.openbabel_types.OpenBabelMACCSFingerprintType_v1* - OpenBabel-MACCS/1 - Open Babel 166 MACCS keys
- *chemfp.openbabel_types.OpenBabelMACCSFingerprintType_v2* - OpenBabel-MACCS/2 - Open Babel 166 MACCS keys
- *chemfp.openbabel_patterns.SubstructOpenBabelFingerprinter_v1* - ChemFP-Substruct-OpenBabel/1 - chemfp's 881 CACTVS/PubChem-like keys implemented with Open Babel

- *chemfp.openbabel_patterns.RDMACCSOpenBabelFingerprinter_v1* - RDMACCS-OpenBabel/1 - chemfp's own 166 MACCS keys implemented with Open Babel (does not include key 44)
- *chemfp.openbabel_patterns.RDMACCSOpenBabelFingerprinter_v2* - RDMACCS-OpenBabel/1 - chemfp's own 166 MACCS keys implemented with Open Babel
- *chemfp.openeye_types.OpenEyeCircularFingerprintType_v2* - OpenEye-Circular/2 - OEGraphSim circular fingerprints
- *chemfp.openeye_types.OpenEyeMACCSFingerprintType_v2* - OpenEye-MACCS166/2 - OEGraphSim 166 MACCS keys
- *chemfp.openeye_types.OpenEyePathFingerprintType_v2* - OpenEye-Path/2 - OEGraphSim path fingerprints
- *chemfp.openeye_types.OpenEyeTreeFingerprintType_v2* - OpenEye-Tree/2 - OEGraphSim tree fingerprints
- *chemfp.openeye_patterns.SubstructOpenEyeFingerprinter_v1* - ChemFP-Substruct-OpenEye/1 - chemfp's 881 CACTVS/PubChem-like keys implemented with OEChem
- *chemfp.openeye_patterns.RDMACCSOpenEyeFingerprinter_v1* - RDMACCS-OpenEye/1 - chemfp's own 166 MACCS keys implemented with OEChem (does not include key 44)
- *chemfp.openeye_patterns.RDMACCSOpenEyeFingerprinter_v2* - RDMACCS-OpenEye/2 - chemfp's own 166 MACCS keys implemented with OEChem
- *chemfp.rdkit_types.RDKitFingerprintType_v1* - RDKit-Fingerprint/1 - RDKit path and tree fingerprint
- *chemfp.rdkit_types.RDKitFingerprintType_v2* - RDKit-Fingerprint/2 - RDKit path and tree fingerprint
- *chemfp.rdkit_types.RDKitMACCSFingerprintType_v1* - RDKit-MACCS/1 - RDKit 166 MACCS keys (does not include key 44)
- *chemfp.rdkit_types.RDKitMACCSFingerprintType_v2* - RDKit-MACCS/2 - RDKit 166 MACCS keys
- *chemfp.rdkit_types.RDKitMorganFingerprintType_v1* - RDKit-Morgan/1 - RDKit circular fingerprints
- *chemfp.rdkit_types.RDKitAtomPairFingerprint_v1* - RDKit-AtomPair/1 - RDKit atom pair fingerprints
- *chemfp.rdkit_types.RDKitAtomPairFingerprint_v2* - RDKit-AtomPair/2 - RDKit atom pair fingerprints
- *chemfp.rdkit_types.RDKitTorsionFingerprintType_v1* - RDKit-Torsion/1 - RDKit torsion fingerprints
- *chemfp.rdkit_types.RDKitTorsionFingerprintType_v2* - RDKit-Torsion/2 - RDKit torsion fingerprints
- chemfp.rdkit_types.RDKitTorsionFingerprintType_v3 - RDKit-Torsion/3 - RDKit torsion fingerprints
- *chemfp.rdkit_patterns.SubstructRDKitFingerprintType_v1* - ChemFP-Substruct-RDKit/1 - chemfp's 881 CACTVS/PubChem-like keys implemented with RDKit

- *chemfp.rdkit_patterns.RDMACCSRDKitFingerprinter_v1* - RDMACCS-RDKit/1 - chemfp's own 166 MACCS keys implemented with OEChem (does not include key 44)
- *chemfp.rdkit_patterns.RDMACCSRDKitFingerprinter_v2* - RDMACCS-RDKit/2 - chemfp's own 166 MACCS keys implemented with OEChem

get_type()
Get the full type string (name and parameters) for this fingerprint type

Returns a canonical fingerprint type string, including its parameters

get_metadata(*sources=None*)
Return a Metadata appropriate for the given fingerprint type.

This is most commonly used to make a *chemfp.Metadata* that can be passed into a *chemfp.FingerprintWriter*.

If *sources* is a string or a list of strings then it will passed to the newly created Metadata instance. It should contain filenames or other description of the fingerprint sources.

Parameters **sources** (*None, a string, or list of strings*) – fingerprint source filenames or other description

Returns a *chemfp.Metadata*

make_fingerprinter()
Make a 'fingerprinter'; a callable which takes a molecule and returns a fingerprint

Returns a function object which takes a molecule and return a fingerprint

read_molecule_fingerprints(*source*, *format=None*, *id_tag=None*, *reader_args=None*, *errors="strict"*, *location=None*)
Read fingerprints from a structure source as a FingerprintIterator

Iterate through the *format* structure records in *source*. If *format* is None then auto-detect the format based on the *source*. Use the fingerprint type to compute the fingerprint. For SD files, use *id_tag* to get the record id from the given SD tag instead of the title line.

The *reader_args* dictionary parameters depend on the toolkit and format. For details see the docstring for `self.toolkit.read_molecules`.

The *errors* parameter specifies how to handle errors. "strict" raises an exception, "report" sends a message to stderr and goes to the next record, and "ignore" goes to the next record.

The *location* parameter takes a Location instance. If None then a default Location will be created.

Parameters

- **source** (*a filename, file object, or None to read from stdin*) – the structure source
- **format** (*a format name string, or Format object, or None to auto-detect*) – the input structure format
- **id_tag** (*string, or None to use the record title*) – SD tag containing the record id
- **reader_args** (*a dictionary*) – reader parameters passed to the underlying toolkit
- **errors** (*one of "strict", "report", or "ignore"*) – specify how to handle errors
- **location** (*a Location object, or None*) – object used to track parser state information

Returns a *chemfp.FingerprintIterator* which iterates over the (id, fingerprint) pair

read_molecule_fingerprints_from_string(*content*, *format=None*, *id_tag=None*, *reader_args=None*, *errors="strict"*, *location=None*)

Read fingerprints from structure records in a string, as a FingerprintIterator

Iterate through the *format* structure records in *content*. Use the fingerprint type to compute the fingerprint. For SD files, use *id_tag* to get the record id from the given SD tag instead of the title line.

The *reader_args* dictionary parameters depend on the toolkit and format. For details see the docstring for `self.toolkit.read_molecules`.

The *errors* parameter specifies how to handle errors. "strict" raises an exception, "report" sends a message to stderr and goes to the next record, and "ignore" goes to the next record.

The *location* parameter takes a Location instance. If None then a default Location will be created.

Parameters

- **content** – the string containing structure records
- **format** (*a format name string, or Format object*) – the input structure format
- **id_tag** (*string, or None to use the record title*) – SD tag containing the record id
- **reader_args** (*a dictionary*) – reader parameters passed to the underlying toolkit
- **errors** (*one of "strict", "report", or "ignore"*) – specify how to handle errors
- **location** (*a Location object, or None*) – object used to track parser state information

Returns a *chemfp.FingerprintIterator* which iterates over the (id, fingerprint) pair

parse_molecule_fingerprint(*content*, *format*, *reader_args=None*, *errors="strict"*)

Parse the first molecule record of the content then compute and return the fingerprint

Read the first molecule from *content*, which contains records in the given *format*. Compute and return its fingerprint.

The *reader_args* dictionary parameters depend on the toolkit and format. For details see the docstring for `self.toolkit.read_molecules`.

The *errors* parameter specifies how to handle errors. "strict" raises an exception, "report" sends a message to stderr and return None for the fingerprint, and "ignore" returns None for the fingerprint without any extra message.

Parameters

- **content** – the string containing at least one structure record
- **format** (*a format name string, or Format object*) – the input structure format
- **reader_args** (*a dictionary*) – reader parameters passed to the underlying toolkit
- **errors** (*one of "strict", "report", or "ignore"*) – specify how to handle errors

Returns the fingerprint as a byte string

parse_id_and_molecule_fingerprint(*content*, *format*, *id_tag=None*, *reader_args=None*, *errors="strict"*)

Parse the first molecule record of the content then compute and return the id and fingerprint

Read the first molecule from *content*, which contains records in the given *format*. Compute its fingerprint and get the molecule id. For an SD record use *id_tag* to get the record id from the given SD tag instead of from the title line.

Return the id and fingerprint as the (id, fingerprint) pair.

The *reader_args* dictionary parameters depend on the toolkit and format. For details see the docstring for `self.toolkit.read_molecules`.

The *errors* parameter specifies how to handle errors. "strict" raises an exception, "report" sends a message to stderr and return None for values it cannot compute, and "ignore" is like "report" but without the error message. For "report" and "ignore", if the molecule cannot be parsed then the result will be (None, None). If the fingerprint cannot be computed then the result will be (id, None).

Parameters

- **content** – the string containing at least one structure record
- **format** (*a format name string, or Format object*) – the input structure format
- **id_tag** (*string, or None to use the record title*) – SD tag containing the record id
- **reader_args** (*a dictionary*) – reader parameters passed to the underlying toolkit
- **errors** (*one of "strict", "report", or "ignore"*) – specify how to handle errors

Returns a pair of (id string, fingerprint byte string)

make_id_and_molecule_fingerprint_parser(*format*, *id_tag=None*, *reader_args=None*, *errors="strict"*)

Make a function which parses molecule from a record and returns the id and computed fingerprint

This is a very specialized function, designed for performance, but it doesn't appear to give any advantage. You likely don't need it.

Return a function which parses a content string containing structure records in the given *format* to get a molecule. Use the molecule to compute the fingerprint and get its id. For an SD record use *id_tag* to get the record id from the given SD tag instead of from the title line.

The new function will return the (id, fingerprint) pair.

The *reader_args* dictionary parameters depend on the toolkit and format. For details see the docstring for `self.toolkit.read_molecules`.

The *errors* parameter specifies how to handle errors. "strict" raises an exception, "report" sends a message to stderr and return None for values it cannot compute, and "ignore" is like "report" but without the error message. For "report" and "ignore", if the molecule cannot be parsed then the result will be (None, None). If the fingerprint cannot be computed then the result will be (id, None).

Parameters

- **format** (*a format name string, or Format object*) – the input structure format
- **id_tag** (*string, or None to use the record title*) – SD tag containing the record id
- **reader_args** (*a dictionary*) – reader parameters passed to the underlying toolkit

- **errors** (*one of "strict", "report", or "ignore"*) – specify how to handle errors

Returns a function which takes a content string and returns an (id, fingerprint) pair

compute_fingerprint(*mol*)
Compute and return the fingerprint byte string for the toolkit molecule

Parameters **mol** – a toolkit molecule

Returns the fingerprint as a byte string

compute_fingerprints(*mols*)
Compute and return the fingerprint for each toolkit molecule in an iterator

This function is a slightly optimized version of:

```
for mol in mols:
  yield self.compute_fingerprint(mol)
```

Parameters **mols** – an iterable of toolkit molecules

Returns a generator of fingerprints, one per molecule

get_fingerprint_family()
Return the fingerprint family for this fingerprint type

Returns a *FingerprintFamily*

Open Babel fingerprints

Open Babel implements four fingerprints families and chemfp implements two fingerprint families using the Open Babel toolkit. These are:

- OpenBabel-FP2 - Indexes linear fragments up to 7 atoms.
- OpenBabel-FP3 - SMARTS patterns specified in the file patterns.txt
- OpenBabel-FP4 - SMARTS patterns specified in the file SMARTS_InteLigand.txt
- OpenBabel-MACCS - SMARTS patterns specified in the file MACCS.txt, which implements nearly all of the 166 MACCS keys
- RDMACCS-OpenBabel - a chemfp implementation of nearly all of the MACCS keys
- ChemFP-Substruct-OpenBabel - an experimental chemfp implementation of the PubChem keys

Most people use FP2 and MACCS.

Note: chemfp-2.0 implements both RDMACCS-OpenBabel/1 and RDMACCS-OpenBabel/2. Version 1 did not have a definition for key 44.

OpenBabelFP2FingerprintType_v1

class chemfp.openbabel_types.**OpenBabelFP2FingerprintType_v1**
OpenBabel FP2 fingerprint based on path enumeration

See http://openbabel.org/wiki/FP2

This is a Daylight-like path enumeration fingerprint with 1021 bits.

The OpenBabel-FP2/1 *FingerprintType* has no parameters.

OpenBabelFP3FingerprintType_v1

class chemfp.openbabel_types.**OpenBabelFP3FingerprintType_v1**
OpenBabel FP3 fingerprint

See http://openbabel.org/wiki/FP3

55 bit fingerprints based on a set of SMARTS patterns defining functional groups.

The OpenBabel-FP3/1 *FingerprintType* has no parameters.

OpenBabelFP4FingerprintType_v1

class chemfp.openbabel_types.**OpenBabelFP4FingerprintType_v1**
OpenBabel FP4 fingerprint

http://openbabel.org/wiki/FP4

307 bit fingerprints based on a set of SMARTS patterns defining functional groups.

The OpenBabel-FP4/1 *FingerprintType* has no parameters.

OpenBabelMACCSFingerprintType_v1

class chemfp.openbabel_types.**OpenBabelMACCSFingerprintType_v1**
Open Babel's implementation of the 166 MACCS keys

WARNING: This implementation contains serious bugs! All of the ring sizes are wrong.

See http://openbabel.org/wiki/Tutorial:Fingerprints and https://github.com/openbabel/openbabel/blob/master/data/MACCS.txt .

The OpenBabel-MACCS/1 *FingerprintType* has no parameters.

Note: this version is only available in older (pre-2012) versions of Open Babel.

OpenBabelMACCSFingerprintType_v2

class chemfp.openbabel_types.**OpenBabelMACCSFingerprintType_v2**
Open Babel's implementation of the 166 MACCS keys

See http://openbabel.org/wiki/Tutorial:Fingerprints and https://github.com/openbabel/openbabel/blob/master/data/MACCS.txt .

Note: Open Babel added support for key 44 on 20 October 2014. This should have been version 3. However, I didn't notice until 1 May 2017 that there was no chemfp test for it. Since everyone has been using it as v2, and very few people used the older version, I won't change the version number.

The OpenBabel-MACCS/2 *FingerprintType* has no parameters.

SubstructOpenBabelFingerprinter_v1

class chemfp.openbabel_patterns.**SubstructOpenBabelFingerprinter_v1**
chemfp's Substruct fingerprint implementation for OEChem, version 1

WARNING: these fingerprints have not been validated.

The Substruct fingerprints are CACTVS/PubChem-like fingerprints designed for use across multiple toolkits.

The ChemFP-Substruct-OpenBabel/1 *FingerprintType* has no parameters.

RDMACCSOpenBabelFingerprinter_v1

class chemfp.openbabel_patterns.**RDMACCSOpenBabelFingerprinter_v1**
chemfp's RDMACCS fingerprint implementation for Open Babel, version 1

The RDMACSS keys are MACCS-166-like fingerprints based on RDKit's MACCS116 definition, but designed to be (slightly) more portable across multiple chemistry toolkits.

This version does not define key 44.

The RDMACSS-OpenBabel/1 *FingerprintType* has no parameters.

RDMACCSOpenBabelFingerprinter_v2

class chemfp.openbabel_patterns.**RDMACCSOpenBabelFingerprinter_v2**
chemfp's RDMACCS fingerprint implementation for Open Babel, version 2

The RDMACSS keys are MACCS-166-like fingerprints based on RDKit's MACCS116 definition, but designed to be (slightly) more portable across multiple chemistry toolkits.

This version defines key 44.

The RDMACSS-OpenBabel/2 *FingerprintType* has no parameters.

OpenEye fingerprints

OpenEye's OEGraphSim library implements four bitstring-based fingerprint families, and chemfp implements two fingerprint families based on OEChem. These are:

- OpenEye-Path - exhaustive enumeration of all linear fragments up to a given size
- OpenEye-Circular - exhaustive enumeration of all circular fragments grown radially from each heavy atom up to a given radius
- OpenEye-Tree - exhaustive enumeration of all trees up to a given size
- OpenEye-MACCS166 - an implementation of the 166 MACCS keys
- RDMACCS-OpenEye - a chemfp implementation of the 166 MACCS keys
- ChemFP-Substruct-OpenEye - an experimental chemfp implementation of the PubChem keys

Note: chemfp-2.0 implements both RDMACCS-OpenEye/1 and RDMACCS-OpenEye/2. Version 1 did not have a definition for key 44.

OpenEyeCircularFingerprintType_v2

class chemfp.openeye_types.**OpenEyeCircularFingerprintType_v2**
OEGraphSim fingerprint based on circular fingerprints around heavy atoms, version 2

See https://docs.eyesopen.com/toolkits/cpp/graphsimtk/fingerprint.html#section-fingerprint-circular

The OpenEye-Circular/2 *FingerprintType* parameters are:

•numbits - the number of bits in the fingerprint (default: 4096)

•minradius - the minimum radius (default: 0)

•maxradius - the maximum radius (default: 5)

•atype - the atom type (default: "Default")

•btype - the bond type (default: "Default")

The atype is either 0 or a 'l' separated string containing one or more of the following: Aromaticity, AtomicNumber, Chiral, EqHBondAcceptor, EqHBondDonor, EqHalogen, FormalCharge, HCount, HvyDegree, Hybridization, InRing, EqAromatic,

The btype is either 0 or a 'l' separated string containing one or more of the following: BondOrder, Chiral, InRing.

OpenEyeMACCSFingerprintType_v2

class chemfp.openeye_types.**OpenEyeMACCSFingerprintType_v2**
OEGraphSim implementation of the 166 MACCS keys, version 2

See https://docs.eyesopen.com/toolkits/cpp/graphsimtk/fingerprint.html#maccs .

The OpenEye-MACCS166/2 *FingerprintType* has no parameters.

This corresponds to GraphSim version '2.0.0'.

OpenEyeMACCSFingerprintType_v3

class chemfp.openeye_types.**OpenEyeMACCSFingerprintType_v3**
OEGraphSim implementation of the 166 MACCS keys, version 3

See https://docs.eyesopen.com/toolkits/cpp/graphsimtk/fingerprint.html#maccs .

The OpenEye-MACCS166/3 *FingerprintType* has no parameters.

This corresponds to GraphSim version '2.2.0', with fixes for bits 91 and 92.

OpenEyePathFingerprintType_v2

class chemfp.openeye_types.**OpenEyePathFingerprintType_v2**
OEGraphSim fingerprint based on path-based enumeration, version 2

See https://docs.eyesopen.com/toolkits/cpp/graphsimtk/fingerprint.html#section-fingerprint-path

The OpenEye-Path/2 *FingerprintType* parameters are:

•numbits - the number of bits in the fingerprint (default: 4096)

•minbonds - the minimum number of bonds (default: 0)

•maxbonds - the maximum number of bonds (default: 5)

•atype - the atom type (default: "Default")

•btype - the bond type (default: "Default")

The atype is either 0 or a 'l' separated string containing one or more of the following: Aromaticity, AtomicNumber, Chiral, EqHBondAcceptor, EqHBondDonor, EqHalogen, FormalCharge, HCount, HvyDegree, Hybridization, InRing, EqAromatic,

The btype is either 0 or a 'l' separated string containing one or more of the following: BondOrder, Chiral, InRing.

OpenEyeTreeFingerprintType_v2

class chemfp.openeye_types.**OpenEyeTreeFingerprintType_v2**
OEGraphSim fingerprint based on tree fingerprints, version 2

See https://docs.eyesopen.com/toolkits/cpp/graphsimtk/fingerprint.html#section-fingerprint-tree

The OpenEye-Tree/2 *FingerprintType* parameters are:

- numbits - the number of bits in the fingerprint (default: 4096)
- minbonds - minimum number of bonds in the tree
- maxbonds - maximum number of bonds in the tree
- atype - the atom type (default: "Default")
- btype - the bond type (default: "Default")

The atype is either 0 or a 'l' separated string containing one or more of the following: Aromaticity, AtomicNumber, Chiral, EqHBondAcceptor, EqHBondDonor, EqHalogen, FormalCharge, HCount, HvyDegree, Hybridization, InRing, EqAromatic,

The btype is either 0 or a 'l' separated string containing one or more of the following: BondOrder, Chiral, InRing.

SubstructOpenEyeFingerprinter_v1

class chemfp.openeye_patterns.**SubstructOpenEyeFingerprinter_v1**
chemfp's Substruct fingerprint implementation for OEChem, version 1

WARNING: these fingerprints have not been validated.

The Substruct fingerprints are CACTVS/PubChem-like fingerprints designed for use across multiple toolkits.

The ChemFP-Substruct-OpenEye/1 *FingerprintType* has no parameters.

RDMACCSOpenEyeFingerprinter_v1

class chemfp.openeye_patterns.**RDMACCSOpenEyeFingerprinter_v1**
chemfp's RDMACCS fingerprint implementation for OEChem, version 1

The RDMACSS keys are MACCS-166-like fingerprints based on RDKit's MACCS116 definition, but designed to be (slightly) more portable across multiple chemistry toolkits.

This version does not define key 44.

The RDMACSS-OpenEye/1 *FingerprintType* has no parameters.

RDMACCSOpenEyeFingerprinter_v2

class chemfp.openeye_patterns.**RDMACCSOpenEyeFingerprinter_v2**
chemfp's RDMACCS fingerprint implementation for OEChem, version 2

The RDMACSS keys are MACCS-166-like fingerprints based on RDKit's MACCS116 definition, but designed to be (slightly) more portable across multiple chemistry toolkits.

This version defines key 44.

The RDMACSS-OpenEye/2 *FingerprintType* has no parameters.

RDKit fingerprints

RDKit implements six fingerprint families, and chemfp implements two fingerprint families based on RDKit. These are:

- RDKit-Fingerprint - exhaustive enumeration of linear and branched trees
- RDKit-MACCS166 - The RDKit implementation of the MACCS keys
- RDKit-Morgan - EFCP-like circular fingerprints
- RDKit-AtomPair - atom pair fingerprints
- RDKit-Torsion - topological-torsion fingerprints
- RDKit-Pattern - substructure screen fingerprint
- RDMACCS-RDKit - a chemfp implementation of the 166 MACCS keys
- ChemFP-Substruct-RDKit - an experimental chemfp implementation of the PubChem keys

Note: chemfp-2.0 implements both RDMACCS-RDKit/1 and RDMACCS-RDKit/2. Version 1 did not have a definition for key 44.

RDKitFingerprintType_v1

class `chemfp.rdkit_types.`**`RDKitFingerprintType_v1`**

RDKit's Daylight-like fingerprint based on linear path and branched tree enumeration, version 1

See http://www.rdkit.org/Python_Docs/rdkit.Chem.rdmolops-module.html#RDKFingerprint

The RDKit-Fingerprint/1 *`FingerprintType`* parameters are:

•fpSize - number of bits in the fingerprint (default: 2048)

•minPath - minimum number of bonds (default: 1)

•maxPath - maximum number of bonds (default: 7)

•nBitsPerHash - number of bits to set for each path hash (default: 2)

•useHs - include information about the number of hydrogens on each atom? (default: True)

Note: this version is only available in older (pre-2014) versions of RDKit

RDKitFingerprintType_v2

class `chemfp.rdkit_types.`**`RDKitFingerprintType_v2`**

RDKit's Daylight-like fingerprint based on linear path and branched tree enumeration, version 2

See http://www.rdkit.org/Python_Docs/rdkit.Chem.rdmolops-module.html#RDKFingerprint

The RDKit-Fingerprint/2 *`FingerprintType`* parameters are:

•fpSize - number of bits in the fingerprint (default: 2048)

•minPath - minimum number of bonds (default: 1)

•maxPath - maximum number of bonds (default: 7)

•nBitsPerHash - number of bits to set for each path hash (default: 2)

•useHs - include information about the number of hydrogens on each atom? (default: True)

RDKitMACCSFingerprintType_v1

class chemfp.rdkit_types.**RDKitMACCSFingerprintType_v1**
RDKit's implementation of the 166 MACCS keys, version 1

See http://rdkit.org/Python_Docs/rdkit.Chem.rdMolDescriptors-module.html#GetMACCSKeysFingerprint

The RDKit-MACCS166/1 fingerprints have no parameters.

This version of RDKit does not support MACCS key 44 ("OTHER").

RDKitMACCSFingerprintType_v2

class chemfp.rdkit_types.**RDKitMACCSFingerprintType_v2**
RDKit's implementation of the 166 MACCS keys, version 2

See http://rdkit.org/Python_Docs/rdkit.Chem.rdMolDescriptors-module.html#GetMACCSKeysFingerprint

The RDKit-MACCS166/1 fingerprints have no parameters. RDKit version added this version in late 2014.

RDKitMorganFingerprintType_v1

class chemfp.rdkit_types.**RDKitMorganFingerprintType_v1**
RDKit Morgan (ECFP-like) fingerprints, version 1

See http://rdkit.org/Python_Docs/rdkit.Chem.rdMolDescriptors-module.html#GetMorganFingerprintAsBitVect

The RDKit-Morgan/1 *FingerprintType* parameters are:

- fpSize - number of bits in the fingerprint (default: 2048)
- radius - radius for the Morgan algorithm (default: 2)
- useFeatures - use chemical-feature invariants (default: 0)
- useChirality - use chirality information (default: 0)
- useBondTypes - include bond type information (default: 1)

RDKitAtomPairFingerprint_v1

class chemfp.rdkit_types.**RDKitAtomPairFingerprint_v1**
RDKit atom pair fingerprints, version 1"

See http://rdkit.org/Python_Docs/rdkit.Chem.rdMolDescriptors-module.html#GetHashedAtomPairFingerprintAsBitVect

The RDKit-AtomPair/1 *FingerprintType* parameters are:

- fpSize - number of bits in the fingerprint (default: 2048)
- minLength - minimum bond count for a pair (default: 1)
- maxLength - maximum bond count for a pair (default: 30)

Note: this version is only available in older (pre-2012) versions of RDKit

RDKitAtomPairFingerprint_v2

class chemfp.rdkit_types.**RDKitAtomPairFingerprint_v2**
RDKit atom pair fingerprints, version 2"

See http://rdkit.org/Python_Docs/rdkit.Chem.rdMolDescriptors-module.html#GetHashedAtomPairFingerprintAsBitVect

The RDKit-AtomPair/2 *FingerprintType* parameters are:

- fpSize - number of bits in the fingerprint (default: 2048)
- minLength - minimum bond count for a pair (default: 1)
- maxLength - maximum bond count for a pair (default: 30)

RDKitTorsionFingerprintType_v1

class chemfp.rdkit_types.**RDKitTorsionFingerprintType_v1**
RDKit torsion fingerprints, version 1

See http://www.rdkit.org/Python_Docs/rdkit.Chem.AtomPairs.Torsions-module.html

An implementation of Topological-torsion fingerprints, as described in: R. Nilakantan, N. Bauman, J. S. Dixon, R. Venkataraghavan; "Topological Torsion: A New Molecular Descriptor for SAR Applications. Comparison with Other Descriptors" JCICS 27, 82-85 (1987).

The RDKit-Torsion/1 *FingerprintType* parameters are:

- fpSize - number of bits in the fingerprint (default: 2048)
- targetSize - number of bonds per torsion (default: 4)

Note: this version is only available in older (pre-2014) versions of RDKit

RDKitTorsionFingerprintType_v2

class chemfp.rdkit_types.**RDKitTorsionFingerprintType_v2**
RDKit torsion fingerprints, version 2

See http://www.rdkit.org/Python_Docs/rdkit.Chem.AtomPairs.Torsions-module.html

An implementation of Topological-torsion fingerprints, as described in: R. Nilakantan, N. Bauman, J. S. Dixon, R. Venkataraghavan; "Topological Torsion: A New Molecular Descriptor for SAR Applications. Comparison with Other Descriptors" JCICS 27, 82-85 (1987).

The RDKit-Torsion/2 *FingerprintType* parameters are:

- fpSize - number of bits in the fingerprint (default: 2048)
- targetSize - number of bonds per torsion (default: 4)

RDKitPatternFingerprint_v1

class chemfp.rdkit_types.**RDKitPatternFingerprint_v1**
RDKit's experimental substructure screen fingerprint, version 1

See http://www.rdkit.org/Python_Docs/rdkit.Chem.rdmolops-module.html#PatternFingerprint

The RDKit-Pattern/1 fingerprint has no parameters.

RDKitPatternFingerprint_v2

class chemfp.rdkit_types.**RDKitPatternFingerprint_v2**
RDKit's experimental substructure screen fingerprint, version 2

See http://www.rdkit.org/Python_Docs/rdkit.Chem.rdmolops-module.html#PatternFingerprint

The RDKit-Pattern/2 fingerprint has no parameters.

RDKitPatternFingerprint_v3

class chemfp.rdkit_types.**RDKitPatternFingerprint_v3**
RDKit's experimental substructure screen fingerprint, version 3

See http://www.rdkit.org/Python_Docs/rdkit.Chem.rdmolops-module.html#PatternFingerprint

The RDKit-Pattern/3 fingerprint has no parameters. This version was released 2017.03.1.

RDKitAvalonFingerprintType_v1

class chemfp.rdkit_types.**RDKitAvalonFingerprintType_v1**
Avalon fingerprints

The Avalon Cheminformatics toolkit is available from https://sourceforge.net/projects/avalontoolkit/ . It is not part of the core RDKit distribution. Instead, RDKit has a compile-time option to download and include it as part of the build process.

The Avalon fingerprint are described in the supplemental information for "QSAR - How Good Is It in Practice? Comparison of Descriptor Sets on an Unbiased Cross Section of Corporate Data Sets", Peter Gedeck, Bernhard Rohde, and Christian Bartels, J. Chem. Inf. Model., 2006, 46 (5), pp 1924-1936, DOI: 10.1021/ci050413p. The supplemental information is available from http://pubs.acs.org/doi/suppl/10.1021/ci050413p

It uses a set of feature classes which "have been fine-tuned to provide good screen-out for the set of substructure queries encounted at Novartis while limiting redundancy." The classes are ATOM_COUNT, ATOM_SYMBOL_PATH, AUGMENTED_ATOM, AUGMENTED_BOND, HCOUNT_PAIR, HCOUNT_PATH, RING_PATH, BOND_PATH, HCOUNT_CLASS_PATH, ATOM_CLASS_PATH, RING_PATTERN, RING_SIZE_COUNTS, DEGREE_PATHS, CLASS_SPIDERS, FEATURE_PAIRS and ALL_PATTERNS.

SubstructRDKitFingerprintType_v1

class chemfp.rdkit_patterns.**SubstructRDKitFingerprintType_v1**
chemfp's Substruct fingerprint implementation for RDKit, version 1

WARNING: these fingerprints have not been validated.

The Substruct fingerprints are CACTVS/PubChem-like fingerprints designed for use across multiple toolkits.

The ChemFP-Substruct-RDKit/1 *FingerprintType* has no parameters.

RDMACCSRDKitFingerprinter_v1

class chemfp.rdkit_patterns.**RDMACCSRDKitFingerprinter_v1**
chemfp's RDMACCS fingerprint implementation for RDKit, version 1

The RDMACSS keys are MACCS-166-like fingerprints based on RDKit's MACCS116 definition, but designed to be (slightly) more portable across multiple chemistry toolkits.

This version does not define key 44.

The RDMACSS-RDKit/1 *FingerprintType* has no parameters.

RDMACCSRDKitFingerprinter_v2

class chemfp.rdkit_patterns.**RDMACCSRDKitFingerprinter_v2**
chemfp's RDMACCS fingerprint implementation for RDKit, version 2

The RDMACSS keys are MACCS-166-like fingerprints based on RDKit's MACCS116 definition, but designed to be (slightly) more portable across multiple chemistry toolkits.

This version defines key 44.

The RDMACSS-RDKit/2 *FingerprintType* has no parameters.

chemfp.arena module

There should be no reason for you to import this module yourself. It contains the *FingerprintArena* implementation. FingerprintArena instances are returns part of the public API but should not be constructed directly.

FingerprintArena

class chemfp.arena.**FingerprintArena**

Store fingerprints in a contiguous block of memory for fast searches

A fingerprint arena implements the *chemfp.FingerprintReader* API.

A fingerprint arena stores all of the fingerprints in a continuous block of memory, so the per-molecule overhead is very low.

The fingerprints can be sorted by popcount, so the fingerprints with no bits set come first, followed by those with 1 bit, etc. If `self.popcount_indices` is a non-empty string then the string contains information about the start and end offsets for all the fingerprints with a given popcount. This information is used for the sublinear search methods.

The public attributes are:

metadata
chemfp.Metadata about the fingerprints

ids
list of identifiers, in index order

Other attributes, which might be subject to change, and which I won't fully explain, are:

- arena - a contiguous block of memory, which contains the fingerprints
- start_padding - number of bytes to the first fingerprint in the block
- end_padding - number of bytes after the last fingerprint in the block
- storage_size - number of bytes used to store a fingerprint
- num_bytes - number of bytes in each fingerprint (must be <= storage_size)

- num_bits - number of bits in each fingerprint
- alignment - the fingerprint alignment
- start - the index for the first fingerprint in the arena/subarena
- end - the index for the last fingerprint in the arena/subarena
- arena_ids - all of the identifiers for the parent arena

The FingerprintArena is its own context manager, but it does nothing on context exit. This is a bug when the FingerprintArena uses a memory-mapped FPB file because there is currently no explicit way to close the file. Only the garbage collector is able to do that.

__len__()
Number of fingerprint records in the FingerprintArena

__getitem__(*i*)
Return the (id, fingerprint) pair at index i

__iter__()
Iterate over the (id, fingerprint) contents of the arena

get_fingerprint_type()
Get the fingerprint type object based on the metadata's type field

This uses `self.metadata.type` to get the fingerprint type string then calls *chemfp.get_fingerprint_type()* to get and return a *chemfp.types.FingerprintType* instance.

This will raise a TypeError if there is no metadata, and a ValueError if the type field was invalid or the fingerprint type isn't available.

Returns a *chemfp.types.FingerprintType*

get_fingerprint(*i*)
Return the fingerprint at index *i*

Raises an IndexError if index *i* is out of range.

get_by_id(*id*)
Given the record identifier, return the (id, fingerprint) pair,

If the *id* is not present then return None.

get_index_by_id(*id*)
Given the record identifier, return the record index

If the *id* is not present then return None.

get_fingerprint_by_id(*id*)
Given the record identifier, return its fingerprint

If the *id* is not present then return None

save(*destination*, *format=None*)
Save the fingerprints to a given destination and format

The output format is based on the *format*. If the format is None then the format depends on the *destination* file extension. If the extension isn't recognized then the fingerprints will be saved in "fps" format.

If the output format is "fps" or "fps.gz" then *destination* may be a filename, a file object, or None; None writes to stdout.

If the output format is "fpb" then *destination* must be a filename.

Parameters

- **destination** (*a filename, file object, or None*) – the output destination
- **format** (*None, "fps", "fps.gz", or "fpb"*) – the output format

Returns None

iter_arenas(*arena_size = 1000*)

Base class for all chemfp objects holding fingerprint records

All FingerprintReader instances have a `metadata` attribute containing a Metadata and can be iteratated over to get the (id, fingerprint) for each record.

copy(*indices=None*, *reorder=None*)

Create a new arena using either all or some of the fingerprints in this arena

By default this create a new arena. The fingerprint data block and ids may be shared with the original arena, which makes this a shallow copy. If the original arena is a slice, or "sub-arena" of an arena, then the copy will allocate new space to store just the fingerprints in the slice and use its own list for the ids.

The *indices* parameter, if not None, is an iterable which contains the indicies of the fingerprint records to copy. Duplicates are allowed, though discouraged.

If *indices* are specified then the default *reorder* value of None, or the value True, will reorder the fingerprints for the new arena by popcount. This improves overall search performance. If *reorder* is False then the new arena will preserve the order given by the indices.

If *indices* are not specified, then the default is to preserve the order type of the original arena. Use `reorder=True` to always reorder the fingerprints in the new arena by popcount, and `reorder=False` to always leave them in the current ordering.

```
>>> import chemfp
>>> arena = chemfp.load_fingerprints("pubchem_queries.fps")
>>> arena.ids[1], arena.ids[5], arena.ids[10], arena.ids[18]
(b'9425031', b'9425015', b'9425040', b'9425033')
>>> len(arena)
19
>>> new_arena = arena.copy(indices=[1, 5, 10, 18])
>>> len(new_arena)
4
>>> new_arena.ids
[b'9425031', b'9425015', b'9425040', b'9425033']
>>> new_arena = arena.copy(indices=[18, 10, 5, 1], reorder=False)
>>> new_arena.ids
[b'9425033', b'9425040', b'9425015', b'9425031']
```

Parameters

- **indices** (*iterable containing integers, or None*) – indicies of the records to copy into the new arena
- **reorder** (*True to reorder, False to leave in input order, None for default action*) – describes how to order the fingerprints

count_tanimoto_hits_fp(*query_fp*, *threshold=0.7*)

Count the fingerprints which are sufficiently similar to the query fingerprint

Return the number of fingerprints in the arena which are at least *threshold* similar to the query fingerprint *query_fp*.

Parameters

- **query_fp** (*byte string*) – query fingerprint
- **threshold** (*float between 0.0 and 1.0, inclusive*) – minimum similarity threshold (default: 0.7)

Returns integer count

threshold_tanimoto_search_fp(*query_fp*, *threshold=0.7*)
Find the fingerprints which are sufficiently similar to the query fingerprint

Find all of the fingerprints in this arena which are at least *threshold* similar to the query fingerprint *query_fp*. The hits are returned as a *SearchResult*, in arbitrary order.

Parameters

- **query_fp** (*byte string*) – query fingerprint
- **threshold** (*float between 0.0 and 1.0, inclusive*) – minimum similarity threshold (default: 0.7)

Returns a *SearchResult*

knearest_tanimoto_search_fp(*query_fp*, *k=3*, *threshold=0.7*)
Find the k-nearest fingerprints which are sufficiently similar to the query fingerprint

Find all of the fingerprints in this arena which are at least *threshold* similar to the query fingerprint, and of those, select the top *k* hits. The hits are returned as a *SearchResult*, sorted from highest score to lowest.

Parameters

- **queries** (a *FingerprintArena*) – query fingerprints
- **threshold** (*float between 0.0 and 1.0, inclusive*) – minimum similarity threshold (default: 0.7)

Returns a *SearchResult*

count_tversky_hits_fp(*query_fp*, *threshold=0.7*, *alpha=1.0*, *beta=1.0*)
Count the fingerprints which are sufficiently similar to the query fingerprint

Return the number of fingerprints in the arena which are at least *threshold* similar to the query fingerprint *query_fp*.

Parameters

- **query_fp** (*byte string*) – query fingerprint
- **threshold** (*float between 0.0 and 1.0, inclusive*) – minimum similarity threshold (default: 0.7)

Returns integer count

threshold_tversky_search_fp(*query_fp*, *threshold=0.7*, *alpha=1.0*, *beta=1.0*)
Find the fingerprints which are sufficiently similar to the query fingerprint

Find all of the fingerprints in this arena which are at least *threshold* similar to the query fingerprint *query_fp*. The hits are returned as a *SearchResult*, in arbitrary order.

Parameters

- **query_fp** (*byte string*) – query fingerprint
- **threshold** (*float between 0.0 and 1.0, inclusive*) – minimum similarity threshold (default: 0.7)

Returns a *SearchResult*

knearest_tversky_search_fp(*query_fp*, *k=3*, *threshold=0.7*, *alpha=1.0*, *beta=1.0*)
Find the k-nearest fingerprints which are sufficiently similar to the query fingerprint

Find all of the fingerprints in this arena which are at least *threshold* similar to the query fingerprint, and of those, select the top *k* hits. The hits are returned as a *SearchResult*, sorted from highest score to lowest.

Parameters

- **queries** (a *FingerprintArena*) – query fingerprints
- **threshold** (*float between 0.0 and 1.0, inclusive*) – minimum similarity threshold (default: 0.7)

Returns a *SearchResult*

chemfp.search module

The following functions and classes are in the chemfp.search module.

There are three main classes of functions. The ones ending with `*_fp` use a query fingerprint to search a target arena. The ones ending with `*_arena` use a query arena to search a target arena. The ones ending with `*_symmetric` use arena to search itself, except that a fingerprint is not tested against itself.

These functions share the same name with very similar functions in the top-level *chemfp* module. My apologies for any confusion. The top-level functions are designed to work with both arenas and iterators as the target. They give a simple search API, and automatically process in blocks, to give a balanced trade-off between performance and response time for the first results.

The functions in this module only work with arena as the target. By default it searches the entire arena before returning. If you want to process portions of the arena then you need to specify the range yourself.

chemfp.search.**count_tanimoto_hits_fp**(*query_fp*, *target_arena*, *threshold=0.7*)
Count the number of hits in *target_arena* at least *threshold* similar to the *query_fp*

Example:

```
query_id, query_fp = chemfp.load_fingerprints("queries.fps")[0]
targets = chemfp.load_fingerprints("targets.fps")
print(chemfp.search.count_tanimoto_hits_fp(query_fp, targets, threshold=0.1))
```

Parameters

- **query_fp** (*a byte string*) – the query fingerprint
- **target_arena** – the target arena
- **threshold** (*float between 0.0 and 1.0, inclusive*) – The minimum score threshold.

Returns an integer count

chemfp.search.**count_tanimoto_hits_arena**(*query_arena*, *target_arena*, *threshold=0.7*)
For each fingerprint in *query_arena*, count the number of hits in *target_arena* at least *threshold* similar to it

Example:

```
queries = chemfp.load_fingerprints("queries.fps")
targets = chemfp.load_fingerprints("targets.fps")
counts = chemfp.search.count_tanimoto_hits_arena(queries, targets, threshold=0.1)
print(counts[:10])
```

The result is implementation specific. You'll always be able to get its length and do an index lookup to get an integer count. Currently it's a ctypes array of longs, but it could be an array.array or Python list in the future.

Parameters

- **query_arena** (a *chemfp.arena.FingerprintArena*) – The query fingerprints.
- **target_arena** (a *chemfp.arena.FingerprintArena*) – The target fingerprints.
- **threshold** (*float between 0.0 and 1.0, inclusive*) – The minimum score threshold.

Returns an array of counts

chemfp.search.**count_tanimoto_hits_symmetric**(*arena*, *threshold=0.7*, *batch_size=100*)

For each fingerprint in the *arena*, count the number of other fingerprints at least *threshold* similar to it

A fingerprint never matches itself.

The computation can take a long time. Python won't check check for a ^C until the function finishes. This can be irritating. Instead, process only *batch_size* rows at a time before checking for a ^C.

Note: the *batch_size* may disappear in future versions of chemfp. I can't detect any performance difference between the current value and a larger value, so it seems rather pointless to have. Let me know if it's useful to keep as a user-defined parameter.

Example:

```
arena = chemfp.load_fingerprints("targets.fps")
counts = chemfp.search.count_tanimoto_hits_symmetric(arena, threshold=0.2)
print(counts[:10])
```

The result object is implementation specific. You'll always be able to get its length and do an index lookup to get an integer count. Currently it's a ctype array of longs, but it could be an array.array or Python list in the future.

Parameters

- **arena** (a *chemfp.arena.FingerprintArena*) – the set of fingerprints
- **threshold** (*float between 0.0 and 1.0, inclusive*) – The minimum score threshold.
- **batch_size** (*integer*) – the number of rows to process before checking for a ^C

Returns an array of counts

chemfp.search.**partial_count_tanimoto_hits_symmetric**(*counts*, *arena*, *threshold=0.7*, *query_start=0*, *query_end=None*, *target_start=0*, *target_end=None*)

Compute a portion of the symmetric Tanimoto counts

For most cases, use *chemfp.search.count_tanimoto_hits_symmetric()* instead of this function!

This function is only useful for thread-pool implementations. In that case, set the number of OpenMP threads to 1.

counts is a contiguous array of integers. It should be initialized to zeros, and reused for successive calls.

The function adds counts for counts[*query_start*:*query_end*] based on computing the upper-triangle portion contained in the rectangle *query_start*:*query_end* and *target_start*:target_end* and using symmetry to fill in the lower half.

You know, this is pretty complicated. Here's the bare minimum example of how to use it correctly to process 10 rows at a time using up to 4 threads:

```
import chemfp
import chemfp.search
from chemfp import futures
import array

chemfp.set_num_threads(1)  # Globally disable OpenMP

arena = chemfp.load_fingerprints("targets.fps")  # Load the fingerprints
n = len(arena)
counts = array.array("i", [0]*n)

with futures.ThreadPoolExecutor(max_workers=4) as executor:
    for row in xrange(0, n, 10):
        executor.submit(chemfp.search.partial_count_tanimoto_hits_symmetric,
                        counts, arena, threshold=0.2,
                        query_start=row, query_end=min(row+10, n))

print(counts)
```

Parameters

- **counts** (*a contiguous block of integer*) – the accumulated Tanimoto counts
- **arena** (a *chemfp.arena.FingerprintArena*) – the fingerprints.
- **threshold** (*float between 0.0 and 1.0, inclusive*) – The minimum score threshold.
- **query_start** (*an integer*) – the query start row
- **query_end** (*an integer, or None to mean the last query row*) – the query end row
- **target_start** (*an integer*) – the target start row
- **target_end** (*an integer, or None to mean the last target row*) – the target end row

Returns None

chemfp.search.**count_tversky_hits_fp**(*query_fp*, *target_arena*, *threshold=0.7*, *alpha=1.0*, *beta=1.0*)

Count the number of hits in *target_arena* least *threshold* similar to the *query_fp* (Tversky)

Example:

```
query_id, query_fp = chemfp.load_fingerprints("queries.fps")[0]
targets = chemfp.load_fingerprints("targets.fps")
print(chemfp.search.count_tversky_hits_fp(query_fp, targets, threshold=0.1))
```

Parameters

- **query_fp** (*a byte string*) – the query fingerprint

- **target_arena** – the target arena
- **threshold** (*float between 0.0 and 1.0, inclusive*) – The minimum score threshold.

Returns an integer count

chemfp.search.**count_tversky_hits_arena**(*query_arena*, *target_arena*, *threshold=0.7*, *alpha=1.0*, *beta=1.0*)

For each fingerprint in *query_arena*, count the number of hits in *target_arena* at least *threshold* similar to it

Example:

```
queries = chemfp.load_fingerprints("queries.fps")
targets = chemfp.load_fingerprints("targets.fps")
counts = chemfp.search.count_tversky_hits_arena(queries, targets, threshold=0.1,
             alpha=0.5, beta=0.5)
print(counts[:10])
```

The result is implementation specific. You'll always be able to get its length and do an index lookup to get an integer count. Currently it's a ctypes array of longs, but it could be an array.array or Python list in the future.

Parameters

- **query_arena** (a *chemfp.arena.FingerprintArena*) – The query fingerprints.
- **target_arena** (a *chemfp.arena.FingerprintArena*) – The target fingerprints.
- **threshold** (*float between 0.0 and 1.0, inclusive*) – The minimum score threshold.

Returns an array of counts

chemfp.search.**count_tversky_hits_symmetric**(*arena*, *threshold=0.7*, *alpha=1.0*, *beta=1.0*, *batch_size=100*)

For each fingerprint in the *arena*, count the number of other fingerprints at least *threshold* similar to it

A fingerprint never matches itself.

The computation can take a long time. Python won't check check for a `^C` until the function finishes. This can be irritating. Instead, process only *batch_size* rows at a time before checking for a `^C`.

Note: the *batch_size* may disappear in future versions of chemfp. I can't detect any performance difference between the current value and a larger value, so it seems rather pointless to have. Let me know if it's useful to keep as a user-defined parameter.

Example:

```
arena = chemfp.load_fingerprints("targets.fps")
counts = chemfp.search.count_tversky_hits_symmetric(
      arena, threshold=0.2, alpha=0.5, beta=0.5)
print(counts[:10])
```

The result object is implementation specific. You'll always be able to get its length and do an index lookup to get an integer count. Currently it's a ctype array of longs, but it could be an array.array or Python list in the future.

Parameters

- **arena** (a *chemfp.arena.FingerprintArena*) – the set of fingerprints
- **threshold** (*float between 0.0 and 1.0, inclusive*) – The minimum score threshold.

- **batch_size** (*integer*) – the number of rows to process before checking for a ^C

Returns an array of counts

chemfp.search.**partial_count_tversky_hits_symmetric**(*counts*, *arena*, *threshold=0.7*, *alpha=1.0*, *beta=1.0*, *query_start=0*, *query_end=None*, *target_start=0*, *target_end=None*)

Compute a portion of the symmetric Tversky counts

For most cases, use *chemfp.search.count_tversky_hits_symmetric()* instead of this function!

This function is only useful for thread-pool implementations. In that case, set the number of OpenMP threads to 1.

counts is a contiguous array of integers. It should be initialized to zeros, and reused for successive calls.

The function adds counts for counts[*query_start*:*query_end*] based on computing the upper-triangle portion contained in the rectangle *query_start*:*query_end* and *target_start*:target_end* and using symmetry to fill in the lower half.

You know, this is pretty complicated. Here's the bare minimum example of how to use it correctly to process 10 rows at a time using up to 4 threads:

```
import chemfp
import chemfp.search
from chemfp import futures
import array

chemfp.set_num_threads(1)  # Globally disable OpenMP

arena = chemfp.load_fingerprints("targets.fps")  # Load the fingerprints
n = len(arena)
counts = array.array("i", [0]*n)

with futures.ThreadPoolExecutor(max_workers=4) as executor:
    for row in xrange(0, n, 10):
        executor.submit(chemfp.search.partial_count_tversky_hits_symmetric,
                        counts, arena, threshold=0.2, alpha=0.5, beta=0.5,
                        query_start=row, query_end=min(row+10, n))

print(counts)
```

Parameters

- **counts** (*a contiguous block of integer*) – the accumulated Tversky counts
- **arena** (a *chemfp.arena.FingerprintArena*) – the fingerprints.
- **threshold** (*float between 0.0 and 1.0, inclusive*) – The minimum score threshold.
- **query_start** (*an integer*) – the query start row
- **query_end** (*an integer, or None to mean the last query row*) – the query end row
- **target_start** (*an integer*) – the target start row
- **target_end** (*an integer, or None to mean the last target row*) – the target end row

Returns None

chemfp.search.**threshold_tanimoto_search_fp**(*query_fp*, *target_arena*, *threshold=0.7*)

Search for fingerprint hits in *target_arena* which are at least *threshold* similar to *query_fp*

The hits in the returned *chemfp.search.SearchResult* are in arbitrary order.

Example:

```
query_id, query_fp = chemfp.load_fingerprints("queries.fps")[0]
targets = chemfp.load_fingerprints("targets.fps")
print(list(chemfp.search.threshold_tanimoto_search_fp(query_fp, targets,␣
↪threshold=0.15)))
```

Parameters

- **query_fp** (*a byte string*) – the query fingerprint
- **target_arena** (a *chemfp.arena.FingerprintArena*) – the target arena
- **threshold** (*float between 0.0 and 1.0, inclusive*) – The minimum score threshold.

Returns a *chemfp.search.SearchResult*

chemfp.search.**threshold_tanimoto_search_arena**(*query_arena*, *target_arena*, *threshold=0.7*)

Search for the hits in the *target_arena* at least *threshold* similar to the fingerprints in *query_arena*

The hits in the returned *chemfp.search.SearchResults* are in arbitrary order.

Example:

```
queries = chemfp.load_fingerprints("queries.fps")
targets = chemfp.load_fingerprints("targets.fps")
results = chemfp.search.threshold_tanimoto_search_arena(queries, targets,␣
↪threshold=0.5)
for query_id, query_hits in zip(queries.ids, results):
    if len(query_hits) > 0:
        print(query_id, "->", ", ".join(query_hits.get_ids()))
```

Parameters

- **query_arena** (a *chemfp.arena.FingerprintArena*) – The query fingerprints.
- **target_arena** (a *chemfp.arena.FingerprintArena*) – The target fingerprints.
- **threshold** (*float between 0.0 and 1.0, inclusive*) – The minimum score threshold.

Returns a *chemfp.search.SearchResults*

chemfp.search.**threshold_tanimoto_search_symmetric**(*arena*, *threshold=0.7*, *include_lower_triangle=True*, *batch_size=100*)

Search for the hits in the *arena* at least *threshold* similar to the fingerprints in the arena

When *include_lower_triangle* is True, compute the upper-triangle similarities, then copy the results to get the full set of results. When *include_lower_triangle* is False, only compute the upper triangle.

The hits in the returned *chemfp.search.SearchResults* are in arbitrary order.

The computation can take a long time. Python won't check check for a ^C until the function finishes. This can be irritating. Instead, process only *batch_size* rows at a time before checking for a ^C.

Note: the *batch_size* may disappear in future versions of chemfp. Let me know if it really is useful for you to have as a user-defined parameter.

Example:

```
arena = chemfp.load_fingerprints("queries.fps")
full_result = chemfp.search.threshold_tanimoto_search_symmetric(arena,␣
↪threshold=0.2)
upper_triangle = chemfp.search.threshold_tanimoto_search_symmetric(
          arena, threshold=0.2, include_lower_triangle=False)
assert sum(map(len, full_result)) == sum(map(len, upper_triangle))*2
```

Parameters

- **arena** (a *chemfp.arena.FingerprintArena*) – the set of fingerprints
- **threshold** (*float between 0.0 and 1.0, inclusive*) – The minimum score threshold.
- **include_lower_triangle** (*boolean*) – if False, compute only the upper triangle, otherwise use symmetry to compute the full matrix
- **batch_size** (*integer*) – the number of rows to process before checking for a ^C

Returns a *chemfp.search.SearchResults*

chemfp.search.**partial_threshold_tanimoto_search_symmetric**(*results*, *arena*, *threshold=0.7*, *query_start=0*, *query_end=None*, *target_start=0*, *target_end=None*, *results_offset=0*)

Compute a portion of the symmetric Tanimoto search results

For most cases, use *chemfp.search.threshold_tanimoto_search_symmetric()* instead of this function!

This function is only useful for thread-pool implementations. In that case, set the number of OpenMP threads to 1.

results is a *chemfp.search.SearchResults* instance which is at least as large as the arena. It should be reused for successive updates.

The function adds hits to results[*query_start*:*query_end*], based on computing the upper-triangle portion contained in the rectangle *query_start*:*query_end* and *target_start*:*target_end*.

It does not fill in the lower triangle. To get the full matrix, call *fill_lower_triangle*.

You know, this is pretty complicated. Here's the bare minimum example of how to use it correctly to process 10 rows at a time using up to 4 threads:

```
import chemfp
import chemfp.search
from chemfp import futures
import array

chemfp.set_num_threads(1)
```

```
arena = chemfp.load_fingerprints("targets.fps")
n = len(arena)
results = chemfp.search.SearchResults(n, n, arena.ids)

with futures.ThreadPoolExecutor(max_workers=4) as executor:
    for row in xrange(0, n, 10):
        executor.submit(chemfp.search.partial_threshold_tanimoto_search_symmetric,
                        results, arena, threshold=0.2,
                        query_start=row, query_end=min(row+10, n))

chemfp.search.fill_lower_triangle(results)
```

The hits in the *chemfp.search.SearchResults* are in arbitrary order.

Parameters

- **results** (a *chemfp.search.SearchResults* instance) – the intermediate search results
- **arena** (a *chemfp.arena.FingerprintArena*) – the fingerprints.
- **threshold** (*float between 0.0 and 1.0, inclusive*) – The minimum score threshold.
- **query_start** (*an integer*) – the query start row
- **query_end** (*an integer, or None to mean the last query row*) – the query end row
- **target_start** (*an integer*) – the target start row
- **target_end** (*an integer, or None to mean the last target row*) – the target end row
- **results_offset** – use results[results_offset] as the base for the results
- **results_offset** – an integer

Returns None

chemfp.search.**fill_lower_triangle**(*results*)

Duplicate each entry of *results* to its transpose

This is used after the symmetric threshold search to turn the upper-triangle results into a full matrix.

Parameters **results** (a *chemfp.search.SearchResults*) – search results

chemfp.search.**threshold_tversky_search_fp**(*query_fp*, *target_arena*, *threshold=0.7*, *alpha=1.0*, *beta=1.0*)

Search for fingerprint hits in *target_arena* which are at least *threshold* similar to *query_fp*

The hits in the returned *chemfp.search.SearchResult* are in arbitrary order.

Example:

```
query_id, query_fp = chemfp.load_fingerprints("queries.fps")[0]
targets = chemfp.load_fingerprints("targets.fps")
print(list(chemfp.search.threshold_tversky_search_fp(
           query_fp, targets, threshold=0.15, alpha=0.5, beta=0.5)))
```

Parameters

- **query_fp** (*a byte string*) – the query fingerprint

- **target_arena** (a *chemfp.arena.FingerprintArena*) – the target arena
- **threshold** (*float between 0.0 and 1.0, inclusive*) – The minimum score threshold.

Returns a *chemfp.search.SearchResult*

chemfp.search.**threshold_tversky_search_arena**(*query_arena*, *target_arena*, *threshold=0.7*, *alpha=1.0*, *beta=1.0*)

Search for the hits in the *target_arena* at least *threshold* similar to the fingerprints in *query_arena*

The hits in the returned *chemfp.search.SearchResults* are in arbitrary order.

Example:

```
queries = chemfp.load_fingerprints("queries.fps")
targets = chemfp.load_fingerprints("targets.fps")
results = chemfp.search.threshold_tversky_search_arena(
              queries, targets, threshold=0.5, alpha=0.5, beta=0.5)
for query_id, query_hits in zip(queries.ids, results):
    if len(query_hits) > 0:
        print(query_id, "->", ", ".join(query_hits.get_ids()))
```

Parameters

- **query_arena** (a *chemfp.arena.FingerprintArena*) – The query fingerprints.
- **target_arena** (a *chemfp.arena.FingerprintArena*) – The target fingerprints.
- **threshold** (*float between 0.0 and 1.0, inclusive*) – The minimum score threshold.

Returns a *chemfp.search.SearchResults*

chemfp.search.**threshold_tversky_search_symmetric**(*arena*, *threshold=0.7*, *alpha=1.0*, *beta=1.0*, *include_lower_triangle=True*, *batch_size=100*)

Search for the hits in the *arena* at least *threshold* similar to the fingerprints in the arena

When *include_lower_triangle* is True, compute the upper-triangle similarities, then copy the results to get the full set of results. When *include_lower_triangle* is False, only compute the upper triangle.

The hits in the returned *chemfp.search.SearchResults* are in arbitrary order.

The computation can take a long time. Python won't check check for a `^C` until the function finishes. This can be irritating. Instead, process only *batch_size* rows at a time before checking for a `^C`

Note: the *batch_size* may disappear in future versions of chemfp. Let me know if it really is useful for you to have as a user-defined parameter.

Example:

```
arena = chemfp.load_fingerprints("queries.fps")
full_result = chemfp.search.threshold_tversky_search_symmetric(
      arena, threshold=0.2, alpha=0.5, beta=0.5)
upper_triangle = chemfp.search.threshold_tversky_search_symmetric(
          arena, threshold=0.2, alpha=0.5, beta=0.5, include_lower_triangle=False)
assert sum(map(len, full_result)) == sum(map(len, upper_triangle))*2
```

Parameters

- **arena** (a *chemfp.arena.FingerprintArena*) – the set of fingerprints
- **threshold** (*float between 0.0 and 1.0, inclusive*) – The minimum score threshold.
- **include_lower_triangle** (*boolean*) – if False, compute only the upper triangle, otherwise use symmetry to compute the full matrix
- **batch_size** (*integer*) – the number of rows to process before checking for a ^C

Returns a *chemfp.search.SearchResults*

chemfp.search.**partial_threshold_tversky_search_symmetric**(*results*, *arena*, *threshold=0.7*, *alpha=1.0*, *beta=1.0*, *query_start=0*, *query_end=None*, *target_start=0*, *target_end=None*, *results_offset=0*)

Compute a portion of the symmetric Tversky search results

For most cases, use *chemfp.search.threshold_tversky_search_symmetric()* instead of this function!

This function is only useful for thread-pool implementations. In that case, set the number of OpenMP threads to 1.

results is a *chemfp.search.SearchResults* instance which is at least as large as the arena. It should be reused for successive updates.

The function adds hits to results[*query_start*:*query_end*], based on computing the upper-triangle portion contained in the rectangle *query_start*:*query_end* and *target_start*:*target_end*.

It does not fill in the lower triangle. To get the full matrix, call *fill_lower_triangle*.

You know, this is pretty complicated. Here's the bare minimum example of how to use it correctly to process 10 rows at a time using up to 4 threads:

```
import chemfp
import chemfp.search
from chemfp import futures
import array

chemfp.set_num_threads(1)

arena = chemfp.load_fingerprints("targets.fps")
n = len(arena)
results = chemfp.search.SearchResults(n, n, arena.ids)

with futures.ThreadPoolExecutor(max_workers=4) as executor:
    for row in xrange(0, n, 10):
        executor.submit(chemfp.search.partial_threshold_tversky_search_symmetric,
                        results, arena, threshold=0.2, alpha=0.5, beta=0.5,
                        query_start=row, query_end=min(row+10, n))

chemfp.search.fill_lower_triangle(results)
```

The hits in the *chemfp.search.SearchResults* are in arbitrary order.

Parameters

- **counts** (*a SearchResults instance*) – the intermediate search results

- **arena** (a *chemfp.arena.FingerprintArena*) – the fingerprints.
- **threshold** (*float between 0.0 and 1.0, inclusive*) – The minimum score threshold.
- **query_start** (*an integer*) – the query start row
- **query_end** (*an integer, or None to mean the last query row*) – the query end row
- **target_start** (*an integer*) – the target start row
- **target_end** (*an integer, or None to mean the last target row*) – the target end row
- **results_offset** – use results[results_offset] as the base for the results
- **results_offset** – an integer

Returns None

chemfp.search.**knearest_tanimoto_search_fp**(*query_fp*, *target_arena*, *k=3*, *threshold=0.7*)

Search for *k*-nearest hits in *target_arena* which are at least *threshold* similar to *query_fp*

The hits in the *chemfp.search.SearchResults* are ordered by decreasing similarity score.

Example:

```
query_id, query_fp = chemfp.load_fingerprints("queries.fps")[0]
targets = chemfp.load_fingerprints("targets.fps")
print(list(chemfp.search.knearest_tanimoto_search_fp(query_fp, targets, k=3,␣
↪threshold=0.0)))
```

Parameters

- **query_fp** (*a byte string*) – the query fingerprint
- **target_arena** (a *chemfp.arena.FingerprintArena*) – the target arena
- **k** (*positive integer*) – the number of nearest neighbors to find.
- **threshold** (*float between 0.0 and 1.0, inclusive*) – The minimum score threshold.

Returns a *chemfp.search.SearchResult*

chemfp.search.**knearest_tanimoto_search_arena**(*query_arena*, *target_arena*, *k=3*, *threshold=0.7*)

Search for the *k* nearest hits in the *target_arena* at least *threshold* similar to the fingerprints in *query_arena*

The hits in the *chemfp.search.SearchResults* are ordered by decreasing similarity score.

Example:

```
queries = chemfp.load_fingerprints("queries.fps")
targets = chemfp.load_fingerprints("targets.fps")
results = chemfp.search.knearest_tanimoto_search_arena(queries, targets, k=3,␣
↪threshold=0.5)
for query_id, query_hits in zip(queries.ids, results):
    if len(query_hits) >= 2:
        print(query_id, "->", ", ".join(query_hits.get_ids()))
```

Parameters

- **query_arena** (a *chemfp.arena.FingerprintArena*) – The query fingerprints.
- **target_arena** (a *chemfp.arena.FingerprintArena*) – The target fingerprints.
- **k** (*positive integer*) – the number of nearest neighbors to find.
- **threshold** (*float between 0.0 and 1.0, inclusive*) – The minimum score threshold.

Returns a *chemfp.search.SearchResults*

chemfp.search.**knearest_tanimoto_search_symmetric**(*arena*, *k=3*, *threshold=0.7*, *batch_size=100*)

Search for the *k*-nearest hits in the *arena* at least *threshold* similar to the fingerprints in the arena

The hits in the *SearchResults* are ordered by decreasing similarity score.

The computation can take a long time. Python won't check check for a ^C until the function finishes. This can be irritating. Instead, process only *batch_size* rows at a time before checking for a ^C.

Note: the *batch_size* may disappear in future versions of chemfp. Let me know if it really is useful for you to keep as a user-defined parameter.

Example:

```
arena = chemfp.load_fingerprints("queries.fps")
results = chemfp.search.knearest_tanimoto_search_symmetric(arena, k=3,␣
↪threshold=0.8)
for (query_id, hits) in zip(arena.ids, results):
    print(query_id, "->", ", ".join(("%s %.2f" % hit) for hit in  hits.get_ids_
↪and_scores()))
```

Parameters

- **arena** (a *chemfp.arena.FingerprintArena*) – the set of fingerprints
- **k** (*positive integer*) – the number of nearest neighbors to find.
- **threshold** (*float between 0.0 and 1.0, inclusive*) – The minimum score threshold.
- **include_lower_triangle** (*boolean*) – if False, compute only the upper triangle, otherwise use symmetry to compute the full matrix
- **batch_size** (*integer*) – the number of rows to process before checking for a ^C

Returns a *chemfp.search.SearchResults*

chemfp.search.**knearest_tversky_search_fp**(*query_fp*, *target_arena*, *k=3*, *threshold=0.7*, *alpha=1.0*, *beta=1.0*)

Search for *k*-nearest hits in *target_arena* which are at least *threshold* similar to *query_fp*

The hits in the *chemfp.search.SearchResults* are ordered by decreasing similarity score.

Example:

```
query_id, query_fp = chemfp.load_fingerprints("queries.fps")[0]
targets = chemfp.load_fingerprints("targets.fps")
print(list(chemfp.search.knearest_tversky_search_fp(
        query_fp, targets, k=3, threshold=0.0, alpha=0.5, beta=0.5)))
```

Parameters

- **query_fp** (*a byte string*) – the query fingerprint
- **target_arena** – the target arena
- **k** (*positive integer*) – the number of nearest neighbors to find.
- **threshold** (*float between 0.0 and 1.0, inclusive*) – The minimum score threshold.

Returns a *chemfp.search.SearchResults*

chemfp.search.**knearest_tversky_search_arena**(*query_arena*, *target_arena*, *k=3*, *threshold=0.7*, *alpha=1.0*, *beta=1.0*)

Search for the *k* nearest hits in the *target_arena* at least *threshold* similar to the fingerprints in *query_arena*

The hits in the *chemfp.search.SearchResults* are ordered by decreasing similarity score.

Example:

```
queries = chemfp.load_fingerprints("queries.fps")
targets = chemfp.load_fingerprints("targets.fps")
results = chemfp.search.knearest_tversky_search_arena(
      queries, targets, k=3, threshold=0.5, alpha=0.5, beta=0.5)
for query_id, query_hits in zip(queries.ids, results):
    if len(query_hits) >= 2:
        print(query_id, "->", ", ".join(query_hits.get_ids()))
```

Parameters

- **query_arena** (a *chemfp.arena.FingerprintArena*) – The query fingerprints.
- **target_arena** (a *chemfp.arena.FingerprintArena*) – The target fingerprints.
- **k** (*positive integer*) – the number of nearest neighbors to find.
- **threshold** (*float between 0.0 and 1.0, inclusive*) – The minimum score threshold.

Returns a *chemfp.search.SearchResults*

chemfp.search.**knearest_tversky_search_symmetric**(*arena*, *k=3*, *threshold=0.7*, *alpha=1.0*, *beta=1.0*, *batch_size=100*)

Search for the *k*-nearest hits in the *arena* at least *threshold* similar to the fingerprints in the arena

The hits in the *SearchResults* are ordered by decreasing similarity score.

The computation can take a long time. Python won't check check for a `^C` until the function finishes. This can be irritating. Instead, process only *batch_size* rows at a time before checking for a `^C`.

Note: the *batch_size* may disappear in future versions of chemfp. Let me know if it really is useful for you to keep as a user-defined parameter.

Example:

```
arena = chemfp.load_fingerprints("queries.fps")
results = chemfp.search.knearest_tversky_search_symmetric(
         arena, k=3, threshold=0.8, alpha=0.5, beta=0.5)
for (query_id, hits) in zip(arena.ids, results):
    print(query_id, "->", ", ".join(("%s %.2f" % hit) for hit in  hits.get_ids_
↪and_scores()))
```

Parameters

- **arena** (a *chemfp.arena.FingerprintArena*) – the set of fingerprints
- **k** (*positive integer*) – the number of nearest neighbors to find.
- **threshold** (*float between 0.0 and 1.0, inclusive*) – The minimum score threshold.
- **include_lower_triangle** (*boolean*) – if False, compute only the upper triangle, otherwise use symmetry to compute the full matrix
- **batch_size** (*integer*) – the number of rows to process before checking for a ^C

Returns a *chemfp.search.SearchResults*

chemfp.search.**contains_fp**(*query_fp*, *target_arena*)

Find the target fingerprints which contain the query fingerprint bits as a subset

A target fingerprint contains a query fingerprint if all of the on bits of the query fingerprint are also on bits of the target fingerprint. This function returns a *chemfp.search.SearchResult* containing all of the target fingerprints in *target_arena* that contain the *query_fp*.

The SearchResult scores are all 0.0.

There is currently no direct way to limit the arena search range. Instead create a subarena by using Python's slice notation on the arena then search the subarena.

Parameters

- **query_fp** (*a byte string*) – the query fingerprint
- **target_arena** (a *chemfp.arena.FingerprintArena*) – The target fingerprints.

Returns a SearchResult instance

chemfp.search.**contains_arena**(*query_arena*, *target_arena*)

Find the target fingerprints which contain the query fingerprints as a subset

A target fingerprint contains a query fingerprint if all of the on bits of the query fingerprint are also on bits of the target fingerprint. This function returns a *chemfp.search.SearchResults* where SearchResults[i] contains all of the target fingerprints in *target_arena* that contain the fingerprint for entry *query_arena* [i].

The SearchResult scores are all 0.0.

There is currently no direct way to limit the arena search range, though you can create and search a subarena by using Python's slice notation.

Parameters

- **query_arena** (a *chemfp.arena.FingerprintArena*) – the query fingerprints
- **target_arena** (a *chemfp.arena.FingerprintArena*) – the target fingerprints

Returns a *chemfp.search.SearchResults* instance, of the same size as query_arena

SearchResults

class chemfp.search.**SearchResults**

Search results for a list of query fingerprints against a target arena

This acts like a list of SearchResult elements, with the ability to iterate over each search results, look them up by index, and get the number of scores.

In addition, there are helper methods to iterate over each hit and to get the hit indicies, scores, and identifiers directly as Python lists, sort the list contents, and more.

__len__()
The number of rows in the SearchResults

__iter__()
Iterate over each SearchResult hit

__getitem__(*i*)
Get the *i*-th SearchResult

shape
Read-only attribute.

the tuple (number of rows, number of columns)

The number of columns is the size of the target arena.

iter_indices()
For each hit, yield the list of target indices

iter_ids()
For each hit, yield the list of target identifiers

iter_scores()
For each hit, yield the list of target scores

iter_indices_and_scores()
For each hit, yield the list of (target index, score) tuples

iter_ids_and_scores()
For each hit, yield the list of (target id, score) tuples

clear_all()
Remove all hits from all of the search results

count_all(*min_score=None*, *max_score=None*, *interval="[]"*)
Count the number of hits with a score between *min_score* and *max_score*

Using the default parameters this returns the number of hits in the result.

The default *min_score* of None is equivalent to -infinity. The default *max_score* of None is equivalent to +infinity.

The *interval* parameter describes the interval end conditions. The default of "[]" uses a closed interval, where min_score <= score <= max_score. The interval "()" uses the open interval where min_score < score < max_score. The half-open/half-closed intervals "(]" and "[)" are also supported.

Parameters

- **min_score** (*a float, or None for -infinity*) – the minimum score in the range.
- **max_score** (*a float, or None for +infinity*) – the maximum score in the range.
- **interval** (*one of "[]", "()", "(]", "[)"*) – specify if the end points are open or closed.

Returns an integer count

cumulative_score_all(*min_score=None*, *max_score=None*, *interval="[]"*)
The sum of all scores in all rows which are between *min_score* and *max_score*

Using the default parameters this returns the sum of all of the scores in all of the results. With a specified range this returns the sum of all of the scores in that range. The cumulative score is also known as the raw score.

The default *min_score* of None is equivalent to -infinity. The default *max_score* of None is equivalent to +infinity.

The *interval* parameter describes the interval end conditions. The default of "[]" uses a closed interval, where min_score <= score <= max_score. The interval "()" uses the open interval where min_score < score < max_score. The half-open/half-closed intervals "(]" and "[)" are also supported.

Parameters

- **min_score** (*a float, or None for -infinity*) – the minimum score in the range.
- **max_score** (*a float, or None for +infinity*) – the maximum score in the range.
- **interval** (*one of "[]", "()", "(]", "[)"*) – specify if the end points are open or closed.

Returns a floating point count

reorder_all(*order="decreasing-score"*)

Reorder the hits for all of the rows based on the requested *order*.

The available orderings are:

•increasing-score - sort by increasing score

•decreasing-score - sort by decreasing score

•increasing-index - sort by increasing target index

•decreasing-index - sort by decreasing target index

•move-closest-first - move the hit with the highest score to the first position

•reverse - reverse the current ordering

Parameters **ordering** (*string*) – the name of the ordering to use

to_csr(*dtype=None*)

Return the results as a SciPy compressed sparse row matrix.

The returned matrix has the same shape as the SearchResult instance and can be passed into, for example, a scikit-learn clustering algorithm.

By default the scores are stored with the *dtype* is "float64".

This method requires that SciPy (and NumPy) be installed.

Parameters **dtype** (*string or NumPy type*) – a NumPy numeric data type

SearchResult

class chemfp.search.**SearchResult**

Search results for a query fingerprint against a target arena.

The results contains a list of hits. Hits contain a target index, score, and optional target ids. The hits can be reordered based on score or index.

__len__()

The number of hits

__iter__()
Iterate through the pairs of (target index, score) using the current ordering

clear()
Remove all hits from this result

get_indices()
The list of target indices, in the current ordering.

get_ids()
The list of target identifiers (if available), in the current ordering

iter_ids()
Iterate over target identifiers (if available), in the current ordering

get_scores()
The list of target scores, in the current ordering

get_ids_and_scores()
The list of (target identifier, target score) pairs, in the current ordering

Raises a TypeError if the target IDs are not available.

get_indices_and_scores()
The list of (target index, score) pairs, in the current ordering

reorder(*ordering="decreasing-score"*)
Reorder the hits based on the requested ordering.

The available orderings are:

- increasing-score - sort by increasing score
- decreasing-score - sort by decreasing score
- increasing-index - sort by increasing target index
- decreasing-index - sort by decreasing target index
- move-closest-first - move the hit with the highest score to the first position
- reverse - reverse the current ordering

Parameters **ordering** (*string*) – the name of the ordering to use

count(*min_score=None*, *max_score=None*, *interval="[]"*)
Count the number of hits with a score between *min_score* and *max_score*

Using the default parameters this returns the number of hits in the result.

The default *min_score* of None is equivalent to -infinity. The default *max_score* of None is equivalent to +infinity.

The *interval* parameter describes the interval end conditions. The default of "[]" uses a closed interval, where min_score <= score <= max_score. The interval "()" uses the open interval where min_score < score < max_score. The half-open/half-closed intervals "(]" and "[)" are also supported.

Parameters

- **min_score** (*a float, or None for -infinity*) – the minimum score in the range.
- **max_score** (*a float, or None for +infinity*) – the maximum score in the range.

- **interval** (*one of "[]", "()", "(]", "[)"*) – specify if the end points are open or closed.

Returns an integer count

cumulative_score(*min_score=None*, *max_score=None*, *interval="[]"*)
The sum of the scores which are between *min_score* and *max_score*

Using the default parameters this returns the sum of all of the scores in the result. With a specified range this returns the sum of all of the scores in that range. The cumulative score is also known as the raw score.

The default *min_score* of None is equivalent to -infinity. The default *max_score* of None is equivalent to +infinity.

The *interval* parameter describes the interval end conditions. The default of "[]" uses a closed interval, where min_score <= score <= max_score. The interval "()" uses the open interval where min_score < score < max_score. The half-open/half-closed intervals "(]" and "[)" are also supported.

Parameters

- **min_score** (*a float, or None for -infinity*) – the minimum score in the range.
- **max_score** (*a float, or None for +infinity*) – the maximum score in the range.
- **interval** (*one of "[]", "()", "(]", "[)"*) – specify if the end points are open or closed.

Returns a floating point value

chemfp.bitops module

The following functions from the chemfp.bitops module provide low-level bit operations on byte and hex fingerprints.

chemfp.bitops.**byte_contains**(*sub_fp*, *super_fp*)
Return 1 if the on bits of *sub_fp* are also 1 bits in *super_fp*, that is, if *super_fp* contains *sub_fp*.

chemfp.bitops.**byte_contains_bit**(*fp*, *bit_index*)
Return True if the the given bit position is on, otherwise False

chemfp.bitops.**byte_difference**(*fp1*, *fp2*)
Return the absolute difference (xor) between the two byte strings, fp1 ^ fp2

chemfp.bitops.**byte_from_bitlist**(*fp*[, *num_bits=1024*])
Convert a list of bit positions into a byte fingerprint, including modulo folding

chemfp.bitops.**byte_hex_tanimoto**(*fp1*, *fp2*)
Compute the Tanimoto similarity between the byte fingerprint *fp1* and the hex fingerprint *fp2*. Return a float between 0.0 and 1.0, or raise a ValueError if *fp2* is not a hex fingerprint

chemfp.bitops.**byte_hex_tversky**(*fp1*, *fp2*, *alpha=1.0*, *beta=1.0*)
Compute the Tversky index between the byte fingerprint *fp1* and the hex fingerprint *fp2*. Return a float between 0.0 and 1.0, or raise a ValueError if *fp2* is not a hex fingerprint

chemfp.bitops.**byte_intersect**(*fp1*, *fp2*)
Return the intersection of the two byte strings, *fp1* & *fp2*

chemfp.bitops.**byte_intersect_popcount**(*fp1*, *fp2*)
Return the number of bits set in the instersection of the two byte fingerprints *fp1* and *fp2*

chemfp.bitops.**byte_popcount**(*fp*)
Return the number of bits set in the byte fingerprint *fp*

chemfp.bitops.**byte_tanimoto**(*fp1*, *fp2*)
Compute the Tanimoto similarity between the two byte fingerprints *fp1* and *fp2*

chemfp.bitops.**byte_to_bitlist**(*bitlist*)
Return a sorted list of the on-bit positions in the byte fingerprint

chemfp.bitops.**byte_tversky**(*fp1*, *fp2*, *alpha=1.0*, *beta=1.0*)
Compute the Tversky index between the two byte fingerprints *fp1* and *fp2*

chemfp.bitops.**byte_union**(*fp1*, *fp2*)
Return the union of the two byte strings, *fp1* | *fp2*

chemfp.bitops.**hex_contains**(*sub_fp*, *super_fp*)
Return 1 if the on bits of sub_fp are also on bits in super_fp, otherwise 0. Return -1 if either string is not a hex fingerprint

chemfp.bitops.**hex_contains_bit**(*fp*, *bit_index*)
Return True if the the given bit position is on, otherwise False.

This function does not validate that the hex fingerprint is actually in hex.

chemfp.bitops.**hex_difference**(*fp1*, *fp2*)
Return the absolute difference (xor) between the two hex strings, *fp1* ^ *fp2*. Raises a ValueError for non-hex fingerprints.

chemfp.bitops.**hex_from_bitlist**(*fp*[, *num_bits=1024*])
Convert a list of bit positions into a hex fingerprint, including modulo folding

chemfp.bitops.**hex_intersect**(*fp1*, *fp2*)
Return the intersection of the two hex strings, *fp1* & *fp2*. Raises a ValueError for non-hex fingerprints.

chemfp.bitops.**hex_intersect_popcount**(*fp1*, *fp2*)
Return the number of bits set in the intersection of the two hex fingerprints *fp1* and *fp2*, or raise a ValueError if either string is a non-hex string

chemfp.bitops.**hex_isvalid**(*s*)
Return 1 if the string *s* is a valid hex fingerprint, otherwise 0

chemfp.bitops.**hex_popcount**(*fp*)
Return the number of bits set in a hex fingerprint *fp*, or -1 for non-hex strings

chemfp.bitops.**hex_tanimoto**(*fp1*, *fp2*)
Compute the Tanimoto similarity between two hex fingerprints. Return a float between 0.0 and 1.0, or raise a ValueError if either string is not a hex fingerprint

chemfp.bitops.**hex_tversky**(*fp1*, *fp2*, *alpha=1.0*, *beta=1.0*)
Compute the Tversky index between two hex fingerprints. Return a float between 0.0 and 1.0, or raise a ValueError if either string is not a hex fingerprint

chemfp.bitops.**hex_to_bitlist**(*bitlist*)
Return a sorted list of the on-bit positions in the hex fingerprint

chemfp.bitops.**hex_union**(*fp1*, *fp2*)
Return the union of the two hex strings, *fp1* | *fp2*. Raises a ValueError for non-hex fingerprints.

chemfp.bitops.**hex_encode**(*s*)
Encode the byte string or ASCII string to hex. Returns a text string.

chemfp.bitops.**hex_encode_as_bytes**(*s*)
Encode the byte string or ASCII string to hex. Returns a byte string.

chemfp.bitops.**hex_decode**(*s*)
Decode the hex-encoded value to a byte string

chemfp.encodings

Decode different fingerprint representations into chemfp form. (Currently only decoders are available. Future released may include encoders.)

The chemfp fingerprints are stored as byte strings, with the bytes in least-significant bit order (bit #0 is stored in the first/left-most byte) and with the bits in most-significant bit order (bit #0 is stored in the first/right-most bit of the first byte).

Other systems use different encodings. These include:

- the '0 and '1' characters, as in '00111101'
- hex encoding, like '3d'
- base64 encoding, like 'SGVsbG8h'
- CACTVS's variation of base64 encoding

plus variations of different LSB and MSB orders.

This module decodes most of the fingerprint encodings I have come across. The fingerprint decoders return a 2-ple of the bit length and the chemfp fingerprint. The bit length is None unless the bit length is known exactly, which currently is only the case for the binary and CACTVS fingerprints. (The hex and other encoders must round the fingerprints up to a multiple of 8 bits.)

chemfp.encodings.**from_binary_lsb**(*text*)
Convert a string like '00010101' (bit 0 here is off) into 'xa8'

The encoding characters '0' and '1' are in LSB order, so bit 0 is the left-most field. The result is a 2-ple of the fingerprint length and the decoded chemfp fingerprint

```
>>> from_binary_lsb('00010101')
(8, b'\xa8')
>>> from_binary_lsb('11101')
(5, b'\x17')
>>> from_binary_lsb('00000000000000010000000000000000')
(29, b'\x00\x80\x00\x00')
>>>
```

chemfp.encodings.**from_binary_msb**(*text*)
Convert a string like '10101000' (bit 0 here is off) into 'xa8'

The encoding characters '0' and '1' are in MSB order, so bit 0 is the right-most field.

```
>>> from_binary_msb(b'10101000')
(8, b'\xa8')
>>> from_binary_msb(b'00010101')
(8, b'\x15')
>>> from_binary_msb(b'00111')
(5, b'\x07')
>>> from_binary_msb(b'00000000000001000000000000000000')
(29, b'\x00\x80\x00\x00')
>>>
```

chemfp.encodings.**from_base64**(*text*)
Decode a base64 encoded fingerprint string

The encoded fingerprint must be in chemfp form, with the bytes in LSB order and the bits in MSB order.

```
>>> from_base64("SGk=")
(None, b'Hi')
>>> from binascii import hexlify
>>> hexlify(from_base64("SGk=")[1])
b'4869'
>>>
```

chemfp.encodings.**from_hex**(*text*)
Decode a hex encoded fingerprint string

The encoded fingerprint must be in chemfp form, with the bytes in LSB order and the bits in MSB order.

```
>>> from_hex(b'10f2')
(None, b'\x10\xf2')
>>>
```

Raises a ValueError if the hex string is not a multiple of 2 bytes long or if it contains a non-hex character.

chemfp.encodings.**from_hex_msb**(*text*)
Decode a hex encoded fingerprint string where the bits and bytes are in MSB order

```
>>> from_hex_msb(b'10f2')
(None, b'\xf2\x10')
>>>
```

Raises a ValueError if the hex string is not a multiple of 2 bytes long or if it contains a non-hex character.

chemfp.encodings.**from_hex_lsb**(*text*)
Decode a hex encoded fingerprint string where the bits and bytes are in LSB order

```
>>> from_hex_lsb(b'102f')
(None, b'\x08\xf4')
>>>
```

Raises a ValueError if the hex string is not a multiple of 2 bytes long or if it contains a non-hex character.

chemfp.encodings.**from_cactvs**(*text*)
Decode a 881-bit CACTVS-encoded fingerprint used by PubChem

```
>>> from_cactvs(b"AAADceB7sQAEAAAAAAAAAAAAAAAAAWAAAAAwAAAAAAAAAAABwAAAHwIYAAAADA"␣
↪+
...              b"rBniwygJJqAACqAyVyVACSBAAhhwIa+CC4ZtgIYCLB0/CUpAhgmADIyYcAgAAO"␣
↪+
...              b"AAAAAAABAAAAAAAAAAIAAAAAAAAAA==")
(881, b'\x07\xde\x8d\x00␣
↪\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x80\x06\x00\x00\x00\x0c\x00\x00\x00\x00\x00\x
↪\x00\x84\xe1@X\x1f\x04\x1df\x1b\x10\x06D\x83\xcb\x0f)
↪%\x10\x06\x19\x00\x13\x93\xe1\x00\x01\x00p\x00\x00\x00\x00\x00\x80\x00\x00\x00\x00\x00\x00\x00
↪')
>>>
```

For format details, see ftp://ftp.ncbi.nlm.nih.gov/pubchem/specifications/pubchem_fingerprints.txt

chemfp.encodings.**from_daylight**(*text*)
Decode a Daylight ASCII fingerprint

```
>>> from_daylight(b"I5Z2MLZgOKRcR...1")
(None, b'PyDaylight')
```

See the implementation for format details.

chemfp.encodings.**from_on_bit_positions**(*text*, *num_bits=1024*, *separator=" "*)
Decode from a list of integers describing the location of the on bits

```
>>> from_on_bit_positions("1 4 9 63", num_bits=32)
(32, b'\x12\x02\x00\x80')
>>> from_on_bit_positions("1,4,9,63", num_bits=64, separator=",")
(64, b'\x12\x02\x00\x00\x00\x00\x00\x80')
```

The text contains a sequence of non-negative integer values separated by the *separator* text. Bit positions are folded modulo num_bits.

This is often used to convert sparse fingerprints into a dense fingerprint.

Note: if you have a list of bit position as integer values then you probably want to use *chemfp.bitops.byte_from_bitlist()*.

chemfp.fps_io module

This module is part of the private API. Do not import it directly.

The function *chemfp.open()* returns an FPSReader if the source is an FPS file. The function *chemfp.open_fingerprint_writer()* returns an FPSWriter if the destination is an FPS file.

FPSReader

class chemfp.fps_io.**FPSReader**

FPS file reader

This class implements the *chemfp.FingerprintReader* API. It is also its own a context manager, which automatically closes the file when the manager exists.

The public attributes are:

metadata
a *chemfp.Metadata* instance with information about the fingerprint type

location
a *chemfp.io.Location* instance with parser location and state information

closed
True if the file is open, else False

The FPSReader.location only tracks the "lineno" variable.

__iter__()
Iterate through the (id, fp) pairs

iter_arenas(*arena_size=1000*)
iterate through *arena_size* fingerprints at a time, as subarenas

Iterate through *arena_size* fingerprints at a time, returned as *chemfp.arena.FingerprintArena* instances. The arenas are in input order and not reordered by popcount.

This method helps trade off between performance and memory use. Working with arenas is often faster than processing one fingerprint at a time, but if the file is very large then you might run out of memory, or get bored while waiting to process all of the fingerprint before getting the first answer.

If *arena_size* is None then this makes an iterator which returns a single arena containing all of the fingerprints.

Parameters **arena_size** (*positive integer, or None*) – The number of fingerprints to put into each arena.

Returns an iterator of *chemfp.arena.FingerprintArena* instances

save(*destination*, *format=None*)

Save the fingerprints to a given destination and format

The output format is based on the *format*. If the format is None then the format depends on the *destination* file extension. If the extension isn't recognized then the fingerprints will be saved in "fps" format.

If the output format is "fps" or "fps.gz" then *destination* may be a filename, a file object, or None; None writes to stdout.

If the output format is "fpb" then *destination* must be a filename.

Parameters

- **destination** (*a filename, file object, or None*) – the output destination
- **format** (*None, "fps", "fps.gz", or "fpb"*) – the output format

Returns None

get_fingerprint_type()

Get the fingerprint type object based on the metadata's type field

This uses `self.metadata.type` to get the fingerprint type string then calls *chemfp.get_fingerprint_type()* to get and return a *chemfp.types.FingerprintType* instance.

This will raise a TypeError if there is no metadata, and a ValueError if the type field was invalid or the fingerprint type isn't available.

Returns a *chemfp.types.FingerprintType*

close()

Close the file

count_tanimoto_hits_fp(*query_fp*, *threshold=0.7*)

Count the fingerprints which are sufficiently similar to the query fingerprint

Return the number of fingerprints in the reader which are at least *threshold* similar to the query fingerprint *query_fp*.

Parameters

- **query_fp** (*byte string*) – query fingerprint
- **threshold** (*float between 0.0 and 1.0, inclusive*) – minimum similarity threshold (default: 0.7)

Returns integer count

count_tanimoto_hits_arena(*queries*, *threshold=0.7*)
Count the fingerprints which are sufficiently similar to each query fingerprint

Returns a list containing a count for each query fingerprint in the *queries* arena. The count is the number of fingerprints in the reader which are at least *threshold* similar to the query fingerprint.

The order of results is the same as the order of the queries.

Parameters

- **queries** (a *FingerprintArena*) – query fingerprints
- **threshold** (*float between 0.0 and 1.0, inclusive*) – minimum similarity threshold (default: 0.7)

Returns list of integer counts, one for each query

count_tversky_hits_fp(*query_fp*, *threshold=0.7*, *alpha=1.0*, *beta=1.0*)
Count the fingerprints which are sufficiently similar to the query fingerprint

Return the number of fingerprints in the reader which are at least *threshold* similar to the query fingerprint *query_fp*.

Parameters

- **query_fp** (*byte string*) – query fingerprint
- **threshold** (*float between 0.0 and 1.0, inclusive*) – minimum similarity threshold (default: 0.7)

Returns integer count

threshold_tanimoto_search_fp(*query_fp*, *threshold=0.7*)
Find the fingerprints which are sufficiently similar to the query fingerprint

Find all of the fingerprints in this reader which are at least *threshold* similar to the query fingerprint *query_fp*. The hits are returned as a *SearchResult*, in arbitrary order.

Parameters

- **query_fp** (*byte string*) – query fingerprint
- **threshold** (*float between 0.0 and 1.0, inclusive*) – minimum similarity threshold (default: 0.7)

Returns a *SearchResult*

threshold_tanimoto_search_arena(*queries*, *threshold=0.7*)
Find the fingerprints which are sufficiently similar to each of the query fingerprints

For each fingerprint in the *queries* arena, find all of the fingerprints in this arena which are at least *threshold* similar. The hits are returned as a *SearchResults*, where the hits in each *SearchResult* is in arbitrary order.

Parameters

- **queries** (a *FingerprintArena*) – query fingerprints
- **threshold** (*float between 0.0 and 1.0, inclusive*) – minimum similarity threshold (default: 0.7)

Returns a *SearchResults*

threshold_tversky_search_fp(*query_fp*, *threshold=0.7*)
Find the fingerprints which are sufficiently similar to the query fingerprint

Find all of the fingerprints in this reader which are at least *threshold* similar to the query fingerprint *query_fp*. The hits are returned as a *SearchResult*, in arbitrary order.

Parameters

- **query_fp** (*byte string*) – query fingerprint
- **threshold** (*float between 0.0 and 1.0, inclusive*) – minimum similarity threshold (default: 0.7)

Returns a *SearchResult*

knearest_tanimoto_search_fp(*query_fp*, *k=3*, *threshold=0.7*)

Find the k-nearest fingerprints which are sufficiently similar to the query fingerprint

Find all of the fingerprints in this reader which are at least *threshold* similar to the query fingerprint, and of those, select the top *k* hits. The hits are returned as a *SearchResult*, sorted from highest score to lowest.

Parameters

- **queries** (a *FingerprintArena*) – query fingerprints
- **threshold** (*float between 0.0 and 1.0, inclusive*) – minimum similarity threshold (default: 0.7)

Returns a *SearchResult*

knearest_tanimoto_search_arena(*queries*, *k=3*, *threshold=0.7*)

Find the k-nearest fingerprints which are sufficiently similar to each of the query fingerprints

For each fingerprint in the *queries* arena, find the fingerprints in this reader which are at least *threshold* similar to the query fingerprint, and of those, select the top *k* hits. The hits are returned as a *SearchResults*, where the hits in each *SearchResult* are sorted by similarity score.

Parameters

- **queries** (a *FingerprintArena*) – query fingerprints
- **threshold** (*float between 0.0 and 1.0, inclusive*) – minimum similarity threshold (default: 0.7)

Returns a *SearchResults*

knearest_tversky_search_fp(*query_fp*, *k=3*, *threshold=0.7*, *alpha=1.0*, *beta=1.0*)

Find the k-nearest fingerprints which are sufficiently similar to the query fingerprint

Find all of the fingerprints in this reader which are at least *threshold* similar to the query fingerprint, and of those, select the top *k* hits. The hits are returned as a *SearchResult*, sorted from highest score to lowest.

Parameters

- **queries** (a *FingerprintArena*) – query fingerprints
- **threshold** (*float between 0.0 and 1.0, inclusive*) – minimum similarity threshold (default: 0.7)

Returns a *SearchResult*

FPSWriter

class chemfp.fps_io.**FPSWriter**

Write fingerprints in FPS format.

This is a subclass of *chemfp.FingerprintWriter*.

Instances have the following attributes:

•metadata - a *chemfp.Metadata* instance

•closed - False when the file is open, else True

•location - a *chemfp.io.Location* instance

An FPSWriter is its own context manager, and will close the output file on context exit.

The Location instance supports the "recno", "output_recno", and "lineno" properties.

write_fingerprint(*id*, *fp*)
Write a single fingerprint record with the given id and fp

Parameters

- **id** (*string*) – the record identifier
- **fp** (*bytes*) – the fingerprint

write_fingerprints(*id_fp_pairs*)
Write a sequence of fingerprint records

Parameters **id_fp_pairs** – An iterable of (id, fingerprint) pairs.

close()
Close the writer

This will set self.closed to False.

chemfp.fpb_io module

This module is part of the private API. Do not import directly.

The function *chemfp.open_fingerprint_writer()* returns an OrderedFPBWriter if the destination is an FPB file and *reorder* is True, or an InputOrderFPBWriter if *reorder* is False.

OrderedFPBWriter

class chemfp.fpb_io.**OrderedFPBWriter**
Fingerprint writer for FPB files where the input fingerprint order is preserved

This is a subclass of *chemfp.FingerprintWriter*.

Instances have the following public attributes:

metadata
a *chemfp.Metadata* instance

closed
False when the file is open, else True

Other attributes (like "alignment", "include_hash", "include_popc", "max_spool_size", and "tmpdir") are undocumented and subject to change in the future. Let me know if they are useful.

An OrderedFPBWriter is also is own context manager, and will close the writer on context exit.

write_fingerprint

class chemfp.fpb_io.**write_fingerprint**
Write a single fingerprint record with the given id and fp to the destination

Parameters

- **id** (*string*) – the record identifier
- **fp** (*bytes*) – the fingerprint

write_fingerprints

class chemfp.fpb_io.**write_fingerprints**
Write a sequence of (id, fingerprint) pairs to the destination

Parameters **id_fp_pairs** – An iterable of (id, fingerprint) pairs.

close

class chemfp.fpb_io.**close**
Close the output writer

InputOrderFPBWriter

class chemfp.fpb_io.**InputOrderFPBWriter**
Fingerprint writer for FPB files which preserves the input fingerprint order

This is a subclass of *chemfp.FingerprintWriter*.

Instances have the following public attributes:

metadata
a *chemfp.Metadata* instance

closed
False when the file is open, else True

Other attributes (like "alignment", "include_hash", "include_popc", "max_spool_size", and "tmpdir") are undocumented and subject to change in the future. Let me know if they are useful.

An InputOrderFPBWriter is also is own context manager, and will close the writer on context exit.

write_fingerprint

class chemfp.fpb_io.**write_fingerprint**
Write a single fingerprint record with the given id and fp to the destination

Parameters

- **id** (*string*) – the record identifier
- **fp** (*bytes*) – the fingerprint

write_fingerprints

class chemfp.fpb_io.**write_fingerprints**
Write a sequence of (id, fingerprint) pairs to the destination

Parameters **id_fp_pairs** – An iterable of (id, fingerprint) pairs.

close

class chemfp.fpb_io.**close**
Close the output writer

This will set self.closed to False

chemfp toolkit API

Open Babel, OEChem and RDKit have different ways to read and write molecules. The chemfp toolkit API is a common wrapper API for structure I/O. The chemfp functions work with native toolkit molecules; chemfp does not have a common molecule API. (For that, use Cinfony.)

While the API is the same across *openbabel_toolkit*, *openbabel_toolkit*, *rdkit_toolkit*, and the *text_toolkit*, there are some differences in how they work. For example, each of the toolkits has it own set of reader and writer arguments. The details are available in the documentation, and this chapter acts as a pointer to the specific toolkit documentation.

name

chemfp.toolkit.**name**

The string "openbabel", "openeye", "rdkit", or "text".

[*openbabel_toolkit*] [*openeye_toolkit*] [*rdkit_toolkit*] [*text_toolkit*]

software

chemfp.toolkit.**software**

A string like "OpenBabel/2.4.1", "OEChem/20170208", "RDKit/2016.09.3" or "chemfp/3.1".

[*openbabel_toolkit*] [*openeye_toolkit*] [*rdkit_toolkit*] [*text_toolkit*]

is_licensed

chemfp.toolkit.**is_licensed**()

[*openbabel_toolkit*] [*openeye_toolkit*] [*rdkit_toolkit*] [*text_toolkit*]

Check if the toolkit is licensed.

get_formats

chemfp.toolkit.**get_formats**(*include_unavailable=False*)

[*openbabel_toolkit*] [*openeye_toolkit*] [*rdkit_toolkit*] [*text_toolkit*]

Return a list of structure formats.

get_input_formats

chemfp.toolkit.**get_input_formats**()

[*openbabel_toolkit*] [*openeye_toolkit*] [*rdkit_toolkit*] [*text_toolkit*]

Return a list of input structure formats.

get_output_formats

chemfp.toolkit.**get_output_formats**()

[*openbabel_toolkit*] [*openeye_toolkit*] [*rdkit_toolkit*] [*text_toolkit*]

Return a list of output structure formats.

get_format

chemfp.toolkit.**get_format**(*format*)

[*openbabel_toolkit*] [*openeye_toolkit*] [*rdkit_toolkit*] [*text_toolkit*]

Get a named format.

get_input_format

chemfp.toolkit.**get_input_format**(*format*)

[*openbabel_toolkit*] [*openeye_toolkit*] [*rdkit_toolkit*] [*text_toolkit*]

Get a named input format.

get_output_format

chemfp.toolkit.**get_output_format**(*format*)

[*openbabel_toolkit*] [*openeye_toolkit*] [*rdkit_toolkit*] [*text_toolkit*]

Get a named output format.

get_input_format_from_source

chemfp.toolkit.**get_input_format_from_source**(*source=None*, *format=None*)

[*openbabel_toolkit*] [*openeye_toolkit*] [*rdkit_toolkit*] [*text_toolkit*]

Get an format given an input source.

get_output_format_from_destination

chemfp.toolkit.**get_output_format_from_destination**(*destination=None*, *format=None*)

[*openbabel_toolkit*] [*openeye_toolkit*] [*rdkit_toolkit*] [*text_toolkit*]

Get an format given an output destination.

read_molecules

chemfp.toolkit.**read_molecules**(*source=None*, *format=None*, *id_tag=None*, *reader_args=None*, *errors="strict"*, *location=None"*)

[*openbabel_toolkit*] [*openeye_toolkit*] [*rdkit_toolkit*] [*text_toolkit*]

Read molecules from a structure file.

read_molecules_from_string

chemfp.toolkit.**read_molecules_from_string**(*content*, *format*, *id_tag=None*, *reader_args=None*, *errors="strict"*, *location=None*)

[*openbabel_toolkit*] [*openeye_toolkit*] [*rdkit_toolkit*] [*text_toolkit*]

Read molecules from structure data stored in a string.

read_ids_and_molecules

chemfp.toolkit.**read_ids_and_molecules**(*source=None*, *format=None*, *id_tag=None*, *reader_args=None*, *errors="strict"*, *location=None*)

[*openbabel_toolkit*] [*openeye_toolkit*] [*rdkit_toolkit*] [*text_toolkit*]

Read ids and molecules from a structure file.

read_ids_and_molecules_from_string

chemfp.toolkit.**read_ids_and_molecules_from_string**(*content*, *format*, *id_tag=None*, *reader_args=None*, *errors="strict"*, *location=None*)

[*openbabel_toolkit*] [*openeye_toolkit*] [*rdkit_toolkit*] [*text_toolkit*]

Read ids and molecules from structure data stored in a string.

make_id_and_molecule_parser

chemfp.toolkit.**make_id_and_molecule_parser**(*format*, *id_tag=None*, *reader_args=None*, *errors="strict"*)

[*openbabel_toolkit*] [*openeye_toolkit*] [*rdkit_toolkit*] [*text_toolkit*]

Make a specialized function which returns the id and molecule given a structure record.

parse_molecule

chemfp.toolkit.**parse_molecule**(*content*, *format*, *id_tag=None*, *reader_args=None*, *errors="strict"*)

[*openbabel_toolkit*] [*openeye_toolkit*] [*rdkit_toolkit*] [*text_toolkit*]

Parse a structure record into a molecule.

parse_id_and_molecule

chemfp.toolkit.**parse_id_and_molecule**(*content*, *format*, *id_tag=None*, *reader_args=None*, *errors="strict"*)

[*openbabel_toolkit*] [*openeye_toolkit*] [*rdkit_toolkit*] [*text_toolkit*]

Parse a structure record into an id and molecule.

create_string

chemfp.toolkit.**create_string**(*mol*, *format*, *id=None*, *writer_args=None*, *errors="strict"*)

[*openbabel_toolkit*] [*openeye_toolkit*] [*rdkit_toolkit*] [*text_toolkit*]

Convert a molecule into a Unicode string containg a structure record.

create_bytes

chemfp.toolkit.**create_bytes**(*mol*, *format*, *id=None*, *writer_args=None*, *errors="strict"*)

[*openbabel_toolkit*] [*openeye_toolkit*] [*rdkit_toolkit*] [*text_toolkit*]

Convert a molecule into a byte string containing a structure record.

open_molecule_writer

chemfp.toolkit.**open_molecule_writer**(*destination=None*, *format=None*, *writer_args=None*, *errors="strict"*, *location=None*)

[*openbabel_toolkit*] [*openeye_toolkit*] [*rdkit_toolkit*] [*text_toolkit*]

Create an output molecule writer, for writing to a file.

open_molecule_writer_to_string

chemfp.toolkit.**open_molecule_writer_to_string**(*format*, *writer_args=None*, *errors="strict"*, *location=None*)

[*openbabel_toolkit*] [*openeye_toolkit*] [*rdkit_toolkit*] [*text_toolkit*]

Create an output molecule writer, for writing to a Unicode string.

open_molecule_writer_to_bytes

chemfp.toolkit.**open_molecule_writer_to_bytes**(*format*, *writer_args=None*, *errors="strict"*, *location=None*)

[*openbabel_toolkit*] [*openeye_toolkit*] [*rdkit_toolkit*] [*text_toolkit*]

Create an output molecule writer, for writing to a byte string.

copy_molecule

chemfp.toolkit.**copy_molecule**(*mol*)

[*openbabel_toolkit*] [*openeye_toolkit*] [*rdkit_toolkit*] [*text_toolkit*]

Make a copy of a toolkit molecule.

add_tag

chemfp.toolkit.**add_tag**(*mol*, *tag*, *value*)

[*openbabel_toolkit*] [*openeye_toolkit*] [*rdkit_toolkit*] [*text_toolkit*]

Add an SD tag to the molecule.

get_tag

chemfp.toolkit.**get_tag**(*mol*, *tag*)

[*openbabel_toolkit*] [*openeye_toolkit*] [*rdkit_toolkit*] [*text_toolkit*]

Get an SD tag for a molecule.

get_tag_pairs

chemfp.toolkit.**get_tag_pairs**()

[*openbabel_toolkit*] [*openeye_toolkit*] [*rdkit_toolkit*] [*text_toolkit*]

Get the list of tag name and tag value pairs.

get_id

chemfp.toolkit.**get_id**(*mol*)

[*openbabel_toolkit*] [*openeye_toolkit*] [*rdkit_toolkit*] [*text_toolkit*]

Get the molecule id.

set_id

chemfp.toolkit.**set_id**(*mol*, *id*)

[*openbabel_toolkit*] [*openeye_toolkit*] [*rdkit_toolkit*] [*text_toolkit*]

Set the molecule id.

chemfp.base_toolkit

The chemfp.base_toolkit module contains a few objects which are shared by the differn toolkit. There should be no reason for you to import the module yourself.

FormatMetadata

The `metadata` attribute of the toolkit readers and writers is a FormatMetadata instance. It contains information about the structure file.

Note that this is **not** the same as the fingerprint *chemfp.Metadata* instance, which contains information about the fingerprint file.

FormatMetadata

class chemfp.base_toolkit.**FormatMetadata**

Information about the reader or writer

The public attributes are:

filename
the source or destination filename, the string "<string>" for string-based I/O, or None if not known

record_format
the normalized record format name. All SMILES formats are "smi", and this does not contain compression information

args
the final reader_args or writer_args, after all processing, and as used by the reader and writer

__repr__()
Return a string like 'FormatMeta(filename="cmpds.sdf.gz", record_format="sdf", args={})'

Toolkit readers

The toolkit readers read from structure files. There are several different variations, depending on the function used to read the file. All of the readers are subclasses of *chemfp.base_toolkit.BaseMoleculeReader*.

Function	Returned reader
chemfp.toolkit.read_molecules()	*chemfp.base_toolkit.MoleculeReader*
chemfp.toolkit.read_molecules_from_string()	*chemfp.base_toolkit.MoleculeReader*
chemfp.toolkit.read_ids_and_molecules()	*chemfp.base_toolkit.IdAndMoleculeReader*
chemfp.toolkit.read_ids_and_molecules_from_string()	*chemfp.base_toolkit.IdAndMoleculeReader*
chemfp.text_toolkit.read_sdf_records()	*chemfp.base_toolkit.RecordReader*
chemfp.text_toolkit.read_sdf_records_from_string()	*chemfp.base_toolkit.RecordReader*
chemfp.text_toolkit.read_sdf_ids_and_records()	*chemfp.base_toolkit.IdAndRecordReader*
chemfp.text_toolkit.read_sdf_ids_and_records_from_string()	*chemfp.base_toolkit.IdAndRecordReader*
chemfp.text_toolkit.read_sdf_ids_and_values()	*chemfp.base_toolkit.IdAndRecordReader*
chemfp.text_toolkit.read_sdf_ids_and_values_from_string()	*chemfp.base_toolkit.IdAndRecordReader*

All of the readers have the same API. The major difference is that some readers return a single object during iteration while the others (those with an "And" in the name) return a pair of objects.

BaseMoleculeReader

class chemfp.base_toolkit.**BaseMoleculeReader**

Base class for the toolkit readers

The public attributes are:

metadata
a *chemfp.base_toolkit.FormatMetadata* instance

location
a *chemfp.io.Location* instance

closed
False if the reader is open, otherwise True

Readers are iterators, so iter(reader) returns itself. next(reader) returns either a single object or a pair of objects depending on reader.

Readers are also a context manager, and call self.close() during exit.

chemfp.base_toolkit.**close**()

Close the reader

If the reader wasn't previously closed then close it. This will set the location properties to their final values, close any files that the reader may have opened, and set `self.closed` to False.

class chemfp.base_toolkit.**MoleculeReader**

Read structures from a file and iterate over the toolkit molecules

The public attributes are:

metadata
a *chemfp.base_toolkit.FormatMetadata* instance

location
a *chemfp.io.Location* instance

closed
False if the reader is open, otherwise True

Note: the toolkit implementation is free to reuse a molecule instead of returning a new one each time.

class chemfp.base_toolkit.**IdAndMoleculeReader**
Read structures from a file and iterate over the (id, toolkit molecule) pairs

The public attributes are:

metadata
a *chemfp.base_toolkit.FormatMetadata* instance

location
a *chemfp.io.Location* instance

closed
False if the reader is open, otherwise True

Note: the toolkit implementation is free to reuse a molecule instead of returning a new one each time.

class chemfp.base_toolkit.**RecordReader**
Read and iterate over records as strings

The public attributes are:

metadata
a *chemfp.base_toolkit.FormatMetadata* instance

location
a *chemfp.io.Location* instance

closed
False if the reader is open, otherwise True

class chemfp.base_toolkit.**IdAndRecordReader**
Read records from file and iterate over the (id, record string) pairs

The public attributes are:

metadata
a *chemfp.base_toolkit.FormatMetadata* instance

location
a *chemfp.io.Location* instance

closed
False if the reader is open, otherwise True

Toolkit writers

The chemfp.open_molecule_writer() function returns a *chemfp.base_toolkit.MoleculeWriter*, and chemfp.open_molecule_writer_to_string() returns a *chemfp.base_toolkit.MoleculeStringWriter*. The two classes implement the *chemfp.base_toolkit.BaseMoleculeWriter* API, and MoleculeWriterToString also implements getvalue().

BaseMoleculeWriter

class chemfp.base_toolkit.**BaseMoleculeWriter**

The base molecule writer API, implemented by *MoleculeWriter* and *MoleculeStringWriter*

The public attributes are:

metadata
a *chemfp.base_toolkit.FormatMetadata* instance

location
a *chemfp.io.Location* instance

closed
False if the reader is open, otherwise True

The writer is a context manager, which calls self.close() when the manager exits.

write_molecule(*mol*)
Write a toolkit molecule

Parameters **mol** (*a toolkit molecule*) – the molecule to write

write_molecules(*mols*)
Write a sequence of molecules

Parameters **mols** (*a toolkit molecule iterator*) – the molecules to write

write_id_and_molecule(*id*, *mol*)
Write an identifier and toolkit molecule

If id is None then the output uses the molecule's own id/title. Specifying the id may modify the molecule's id/title, depending on the format and toolkit.

Parameters

- **id** (*string, or None*) – the identifier to use for the molecule
- **mol** (*a toolkit molecule*) – the molecule to write

write_ids_and_molecules(*ids_and_mols*)
Write a sequence of (id, molecule) pairs

This function works well with *chemfp.toolkit.read_ids_and_molecules()*, for example, to convert an SD file to SMILES file, and use an alternate *id_tag* to specify an alternative identifier.

Parameters **mols** (*a (id string, toolkit molecule) iterator*) – the molecules to write

close()
Close the writer

If the reader wasn't previously closed then close it. This will set the location properties to their final values, close any files that the writer may have opened, and set `self.closed` to False.

class chemfp.base_toolkit.**MoleculeWriter**
A BaseMoleculeWriter which writes molecules to a file.

The public attributetes are:

metadata
a *chemfp.base_toolkit.FormatMetadata* instance

location
a *chemfp.io.Location* instance

closed
False if the reader is open, otherwise True

The writer is a context manager, which calls self.close() when the manager exits.

class chemfp.base_toolkit.**MoleculeStringWriter**

A BaseMoleculeWriter which writes molecules to a string.

This class implements the *chemfp.base_toolkit.BaseMoleculeWriter* API.

metadata
a *chemfp.base_toolkit.FormatMetadata* instance

location
a *chemfp.io.Location* instance

closed
False if the reader is open, otherwise True

The writer is a context manager, which calls self.close() when the manager exits.

getvalue()
Get the string containing all of the written record.

This function can also be called after the writer is closed.

Returns a string

Format

Format

class chemfp.base_toolkit.**Format**

Information about a toolkit format.

Use *chemfp.toolkit.get_format()* and related functions to return a Format instance.

The public properties are:

__repr__()
Return a string like 'Format("openeye/sdf.gz")'

prefix
Read-only attribute.

Return the prefix to turn an unqualified parameter into a fully qualified parameter

Returns a string like "rdkit.smi" or "openbabel.sdf"

is_input_format
Read-only attribute.

Return True if this toolkit can read molecules in this format

is_output_format
Read-only attribute.

Return True if this toolkit can write molecules in this format

is_available
Read-only attribute.

Return True if this version of the toolkit understands this format

For example, if your version of RDKit does not support InChI then this would return False for the "inchi" and "inchikey" formats.

supports_io
Read-only attribute.

Return True if this format support reading or writing records

This will return False for formats like "smistring" and "inchikeystring" because those are are not record-based formats.

Note: I don't like this name. I may change it to `is_record_format`. Let me know if you have ideas, or if changing the name will be a problem.

get_reader_args_from_text_settings(*reader_settings*)
Process the *reader_settings* and return the *reader_args* for this format.

This function exists to help convert string settings, eg, from the command-line or a configuration, into usable *reader_args*.

Setting names may be fully-qualified names like "rdkit.sdf.sanitize", partially qualified names like "rdkit.*.sanitize" or "openeye.smi.delimiter", or unqualified names like "delimiter". The qualifiers act as a namespace so the settings can be specified without needing to know the actual toolkit or format.

The function turns the format-appropriate qualified names into unqualified ones and converts the string values into usable Python objects. For example:

```
>>> from chemfp import rdkit_toolkit  as T
>>> fmt = T.get_format("smi")
>>> fmt.get_reader_args_from_text_settings({"rdkit.*.sanitize": "true",
↪"delimiter": "to-eol"})
{'delimiter': 'to-eol', 'sanitize': True}
```

Parameters **reader_settings** (*a dictionary with string keys and values*) – the reader settings

Returns a dictionary of unqualified argument names as keys and processed Python values as values

get_writer_args_from_text_settings(*writer_settings*)
Process *writer_settings* and return the *writer_args* for this format.

This function exists to help convert string settings, eg, from the command-line or a configuration, into usable *writer_args*.

Setting names may be fully-qualified names like "rdkit.sdf.kekulize", partially qualified names like "rdkit.*.delimiter" or "openeye.smi.delimiter", or unqualified names like "delimiter". The qualifiers act as a namespace so the settings can be specified without needing to know the actual toolkit or format.

The function turns the format-appropriate qualified names into unqualified ones and converts the string values into usable Python objects. For example:

```
>>> from chemfp import rdkit_toolkit  as T
>>> fmt = T.get_format("smi")
```

```
>>> fmt.get_writer_args_from_text_settings({"rdkit.*.kekuleSmiles": "true",
↪"canonical": "false"})
{'kekuleSmiles': True, 'canonical': False}
```

Parameters **writer_settings** (*a dictionary with string keys and values*) – the writer settings

Returns a dictionary of unqualified argument names as keys and processed Python values as values

get_default_reader_args()

Return a dictionary of the default reader arguments

The keys are unqualified (ie, without dots).

```
>>> from chemfp import openbabel_toolkit as T
>>> fmt = T.get_format("smi")
>>> fmt.get_default_reader_args()
{'has_header': False, 'delimiter': None, 'options': None}
```

Returns a dictionary of string keys and Python objects for values

get_default_writer_args()

Return a dictionary of the default writer arguments

The keys are unqualified (ie, without dots).

```
>>> from chemfp import openbabel_toolkit as T
>>> fmt = T.get_format("smi")
>>> fmt.get_default_writer_args()
{'explicit_hydrogens': False, 'isomeric': True, 'delimiter': None,
'options': None, 'canonicalization': 'default'}
```

Returns a dictionary of string keys and Python objects for values

get_unqualified_reader_args(*reader_args*)

Convert possibly qualified reader args into unqualified reader args for this format

The *reader_args* dictionary can be confusing because of the priority rules in how to resolve qualifiers, and because it can include irrelevant parameters, which are ignored.

The get_unqualified_reader_args function applies the qualifier resolution algorithm and removes irrelevant parameters to return a dictionary containing the equivalent unqualified reader args dictionary for this format.

```
>>> from chemfp import rdkit_toolkit as T
>> fmt = T.get_format("smi")
>>> fmt.get_unqualified_reader_args({"rdkit.*.delimiter": "tab", "smi.sanitize
↪": False, "X": "Y"})
{'delimiter': 'tab', 'has_header': False, 'sanitize': False}
>>> fmt = T.get_format("can")
>>> fmt.get_unqualified_reader_args({"rdkit.*.delimiter": "tab", "smi.sanitize
↪": False, "X": "Y"})
{'delimiter': 'tab', 'has_header': False, 'sanitize': True}
```

Parameters reader_args reader arguments, which can contain qualified and unqualified arguments

Returns a dictionary of reader arguments, containing only unqualified arguments appropriate for this format.

get_unqualified_writer_args(*writer_args*)
Convert possibly qualified writer args into unqualified writer args for this format

The *writer_args* dictionary can be confusing because of the priority rules in how to resolve qualifiers, and because it can include irrelevant parameters, which are ignored.

The get_unqualified_writer_args function applies the qualifier resolution algorithm and removes irrelevant parameters to return a dictionary containing the equivalent unqualified writer args dictionary for this format.

```
>>> from chemfp import rdkit_toolkit as T
>>> fmt = T.get_format("smi")
>>> fmt.get_unqualified_writer_args({"rdkit.*.delimiter": "tab", "smi.
↪kekuleSmiles": True, "X": "Y"})
{'isomericSmiles': True, 'delimiter': 'tab', 'kekuleSmiles': True,
↪'allBondsExplicit': False, 'canonical': True}
>>> fmt = T.get_format("can")
>>> fmt.get_unqualified_writer_args({"rdkit.*.delimiter": "tab", "smi.
↪kekuleSmiles": True, "X": "Y"})
{'isomericSmiles': False, 'delimiter': 'tab', 'kekuleSmiles': False,
↪'allBondsExplicit': False, 'canonical': True}
```

Parameters writer_args writer arguments, which can contain qualified and unqualified arguments

Returns a dictionary of writer arguments, containing only unqualified arguments appropriate for this format.

chemfp.openbabel_toolkit module

The chemfp toolkit layer for Open Babel.

name

chemfp.openbabel_toolkit.**name**

The string “openbabel”.

software

chemfp.openbabel_toolkit.**software**

A string like “OpenBabel/2.4.1”, where the second part of the string comes from OBReleaseVersion.

is_licensed (openbabel_toolkit)

chemfp.openbabel_toolkit.**is_licensed**()
Return True - Open Babel is always licensed

Returns True

get_formats (openbabel_toolkit)

chemfp.openbabel_toolkit.**get_formats**(*include_unavailable=False*)
Get the list of structure formats that Open Babel supports

If *include_unavailable* is True then also include Open Babel formats which aren't available to this specific version of Open Babel.

Parameters **include_unavailable** (*True or False*) – include unavailable formats?

Returns a list of *chemfp.base_toolkit.Format* objects

get_input_formats (openbabel_toolkit)

chemfp.openbabel_toolkit.**get_input_formats**()
Get the list of supported Open Babel input formats

Returns a list of *chemfp.base_toolkit.Format* objects

get_output_formats (openbabel_toolkit)

chemfp.openbabel_toolkit.**get_output_formats**()
Get the list of supported Open Babel output formats

Returns a list of *chemfp.base_toolkit.Format* objects

get_format (openbabel_toolkit)

chemfp.openbabel_toolkit.**get_format**(*format_name*)
Get the named format, or raise a ValueError

This will raise a ValueError if Open Babel does not implement the format *format_name* or that format is not available.

Parameters **format_name** (*a string*) – the format name

Returns a *chemfp.base_toolkit.Format* object

get_input_format (openbabel_toolkit)

chemfp.openbabel_toolkit.**get_input_format**(*format_name*)
Get the named input format, or raise a ValueError

This will raise a ValueError if Open Babel does not implement the format *format_name* or that format is not an input format.

Parameters **format_name** (*a string*) – the format name

Returns a *chemfp.base_toolkit.Format* object

get_output_format (openbabel_toolkit)

chemfp.openbabel_toolkit.**get_output_format**(*format_name*)

Get the named format, or raise a ValueError

This will raise a ValueError if Open Babel does not implement the format *format_name* or that format is not an output format.

Parameters **format_name** (*a string*) – the format name

Returns a *chemfp.base_toolkit.Format* object

get_input_format_from_source (openbabel_toolkit)

chemfp.openbabel_toolkit.**get_input_format_from_source**(*source=None*, *format=None*)

Get the most appropriate format given the available source and format information

If *format* is a *chemfp.base_toolkit.Format* then return it. If it's a Format-like object with "name" and "compression" attributes use it to make a real Format object with the same attributes. If it's a string then use it to create a Format object.

If *format* is None, use the *source* to auto-detect the format. If auto-detection is not possible, assume it's an uncompressed SMILES file.

Parameters

- **source** (*a filename (as a string), a file object, or None to read from stdin*) – the structure data source.
- **format** (*a Format(-like) object, string, or None*) – format information, if known.

Returns a *chemfp.base_toolkit.Format* object

get_output_format_from_destination (openbabel_toolkit)

chemfp.openbabel_toolkit.**get_output_format_from_destination**(*destination=None*, *format=None*)

Get the most appropriate format given the available destination and format information

If *format* is a *chemfp.base_toolkit.Format* then return it. If it's a Format-like object with "name" and "compression" attributes use it to make a real Format object with the same attributes. If it's a string then use it to create a Format object.

If *format* is None, use the *destination* to auto-detect the format. If auto-detection is not possible, assume it's an uncompressed SMILES file.

Parameters

- **destination** (*a filename (as a string), a file object, or None to read from stdin*) – the structure data source.
- **format** (*a Format(-like) object, string, or None*) – format information, if known.

Returns a *chemfp.base_toolkit.Format* object

read_molecules (openbabel_toolkit)

chemfp.openbabel_toolkit.**read_molecules**(*source=None*, *format=None*, *id_tag=None*, *reader_args=None*, *errors="strict"*, *location=None*, *encoding="utf8"*, *encoding_errors="strict"*)

Return an iterator that reads OBMol molecules from a structure file

Iterate through the *format* structure records in *source*. If *format* is None then auto-detect the format based on the *source*. For SD files, use *id_tag* to get the record id from the given SD tag instead of the title line. (read_molecules() will ignore the *id_tag*. It exists to make it easier to switch between reader functions.)

Note: the reader will clear and reuse the OBMol instance. Make a copy if you want to keep the molecule around.

The *reader_args* dictionary parameters depend on the format. Every Open Babel format supports an "options" entry, which is passed to SetOptions(). See that documentation for details. Some formats support additional parameters:

- SMILES and InChI
  - delimiter - one of "tab", "space", "to-eol", the space or tab characters, or None
  - has_header - True or False
- SDF
  - implementation - if "openbabel" or None, use the Open Babel record parser; if "chemfp", use chemfp's own record parser, which has better location tracking

The *errors* parameter specifies how to handle errors. "strict" raises an exception, "report" sends a message to stderr and goes to the next record, and "ignore" goes to the next record.

The *location* parameter takes a *chemfp.io.Location* instance. If None then a default Location will be created.

See *chemfp.openbabel_toolkit.read_ids_and_molecules()* if you want (id, OBMol) pairs instead of just the molecules.

Parameters

- **source** (*a filename, file object, or None to read from stdin*) – the structure source
- **format** (*a format name string, or Format object, or None to auto-detect*) – the input structure format
- **id_tag** (*string, or None to use the record title*) – SD tag containing the record id
- **reader_args** (*a dictionary*) – reader arguments passed to the underlying toolkit
- **errors** (*one of "strict", "report", or "ignore"*) – specify how to handle errors
- **location** (a *chemfp.io.Location* object, or None) – object used to track parser state information

Returns a *chemfp.base_toolkit.MoleculeReader* iterating OBMol molecules

read_molecules_from_string (openbabel_toolkit)

chemfp.openbabel_toolkit.**read_molecules_from_string**(*content*, *format*, *id_tag=None*, *reader_args=None*, *errors="strict"*, *location=None*)

Return an iterator that reads OBMol molecules from a string containing structure records

content is a string containing 0 or more records in the format *format*. See *chemfp.openbabel_toolkit.read_molecules()* for details about the other parameters. See *chemfp.openbabel_toolkit.read_ids_and_molecules_from_string()* if you want to read (id, OBMol) pairs instead of just molecules.

Note: the reader will clear and reuse the OBMol instance. Make a copy if you want to keep the molecule around.

Parameters

- **content** (*a string*) – the string containing structure records
- **format** (*a format name string, or Format object*) – the input structure format
- **id_tag** (*string, or None to use the record title*) – SD tag containing the record id
- **reader_args** (*a dictionary*) – reader arguments passed to the underlying toolkit
- **errors** (*one of "strict", "report", or "ignore"*) – specify how to handle errors
- **location** (a *chemfp.io.Location* object, or None) – object used to track parser state information

Returns a *chemfp.base_toolkit.MoleculeReader* iterating OBMol molecules

read_ids_and_molecules (openbabel_toolkit)

chemfp.openbabel_toolkit.**read_ids_and_molecules**(*source=None*, *format=None*, *id_tag=None*, *reader_args=None*, *errors="strict"*, *location=None*, *encoding="utf8"*, *encoding_errors="strict"*)

Return an iterator that reads (id, OBMol molecule) pairs from a structure file

See *chemfp.openbabel_toolkit.read_molecules()* for full parameter details. The major difference is that this returns an iterator of (id, OBMol) pairs instead of just the molecules.

Note: the reader will clear and reuse the OBMol instance. Make a copy if you want to keep the molecule around.

Parameters

- **source** (*a filename, file object, or None to read from stdin*) – the structure source
- **format** (*a format name string, or Format object, or None to auto-detect*) – the input structure format
- **id_tag** (*string, or None to use the record title*) – SD tag containing the record id
- **reader_args** (*a dictionary*) – reader arguments passed to the underlying toolkit
- **errors** (*one of "strict", "report", or "ignore"*) – specify how to handle errors
- **location** (a *chemfp.io.Location* object, or None) – object used to track parser state information

Returns a *chemfp.base_toolkit.IdAndMoleculeReader* iterating (id, OBMol) pairs

read_ids_and_molecules_from_string (openbabel_toolkit)

chemfp.openbabel_toolkit.**read_ids_and_molecules_from_string**(*content*, *format*, *id_tag=None*, *reader_args=None*, *errors="strict"*, *location=None*)

Return an iterator that reads (id, OBMol) pairs from a string containing structure records

content is a string containing 0 or more records in the format *format*. See *chemfp.openbabel_toolkit.read_molecules()* for details about the other parameters. See *chemfp.openbabel_toolkit.read_molecules_from_string()* if you just want to read the OBMol molecules instead of (id, OBMol) pairs.

Note: the reader will clear and reuse the OBMol instance. Make a copy if you want to keep the molecule around.

Parameters

- **content** (*a string*) – the string containing structure records
- **format** (*a format name string, or Format object*) – the input structure format
- **id_tag** (*string, or None to use the record title*) – SD tag containing the record id
- **reader_args** (*a dictionary*) – reader arguments passed to the underlying toolkit
- **errors** (*one of "strict", "report", or "ignore"*) – specify how to handle errors
- **location** (a *chemfp.io.Location* object, or None) – object used to track parser state information

Returns a *chemfp.base_toolkit.IdAndMoleculeReader* iterating (id, OBMol) pairs

make_id_and_molecule_parser (openbabel_toolkit)

chemfp.openbabel_toolkit.**make_id_and_molecule_parser**(*format*, *id_tag=None*, *reader_args=None*, *errors="strict"*)

Create a specialized function which takes a record and returns an (id, OBMol) pair

The returned function is optimized for reading many records from individual strings because it only does parameter validation once. The function will reuse the OBMol for successive calls, so make a copy if you want to keep it around. However, I haven't really noticed much of a performance difference between this and *chemfp.openbabel_toolkit.parse_id_and_molecule()* so I suggest you use that function directly instead of making a specialized function. (Let me know if making a specialized function is useful.)

See *chemfp.openbabel_toolkit.read_molecules()* for details about the other parameters.

Parameters

- **format** (*a format name string, or Format object*) – the input structure format
- **id_tag** (*string, or None to use the record title*) – SD tag containing the record id
- **reader_args** (*a dictionary*) – reader arguments passed to the underlying toolkit
- **errors** (*one of "strict", "report", or "ignore"*) – specify how to handle errors

Returns a function of the form `parser(record string) -> (id, OBMol)`

parse_molecule (openbabel_toolkit)

chemfp.openbabel_toolkit.**parse_molecule**(*content*, *format*, *id_tag=None*, *reader_args=None*, *errors="strict"*)

Parse the first structure record from the *content* string and return an OBMol molecule.

content is a string containing a single structure record in format *format*. (Additional records are ignored). See *chemfp.openbabel_toolkit.read_molecules()* for details about the other parameters. See *chemfp.openbabel_toolkit.parse_id_and_molecule()* if you want the (id, OBMol) pair instead of just the molecule.

Parameters

- **content** (*a string*) – the string containing a structure record
- **format** (*a format name string, or Format object*) – the input structure format
- **id_tag** (*string, or None to use the record title*) – SD tag containing the record id
- **reader_args** (*a dictionary*) – reader arguments passed to the underlying toolkit

- **errors** (*one of "strict", "report", or "ignore"*) – specify how to handle errors

Returns an OBMol molecule

parse_id_and_molecule (openbabel_toolkit)

chemfp.openbabel_toolkit.**parse_id_and_molecule**(*content*, *format*, *id_tag=None*, *reader_args=None*, *errors="strict"*)

Parse the first structure record from *content* and return the (id, OBMol) pair.

content is a string containing a single structure record in format *format*. (Additional records are ignored). See *chemfp.openbabel_toolkit.read_molecules()* for details about the other parameters.

See *chemfp.openbabel_toolkit.read_molecules()* for details about the other parameters. See *chemfp.openbabel_toolkit.parse_molecule()* if just want the OBMol molecule and not the the (id, OBMol) pair.

Parameters

- **content** (*a string*) – the string containing a structure record
- **format** (*a format name string, or Format object*) – the input structure format
- **id_tag** (*string, or None to use the record title*) – SD tag containing the record id
- **reader_args** (*a dictionary*) – reader arguments passed to the underlying toolkit
- **errors** (*one of "strict", "report", or "ignore"*) – specify how to handle errors

Returns an (id, OBMol molecule) pair

create_string (openbabel_toolkit)

chemfp.openbabel_toolkit.**create_string**(*mol*, *format*, *id=None*, *writer_args=None*, *errors="strict"*)

Convert an OBMol into a structure record in the given format as a Unicode string

If *id* is not None then use it instead of the molecule's own title. Warning: this may briefly modify the molecule, so may not be thread-safe.

Parameters

- **mol** (*an Open Babel molecule*) – the molecule to use for the output
- **format** (*a format name string, or Format object*) – the output structure format
- **id** (*a string, or None to use the molecule's own id*) – an alternate record id
- **writer_args** (*a dictionary*) – writer arguments passed to the underlying toolkit

- **errors** (*one of "strict", "report", or "ignore"*) – specify how to handle errors

Returns a Unicode string

create_bytes (openbabel_toolkit)

chemfp.openbabel_toolkit.**create_bytes**(*mol*, *format*, *id=None*, *writer_args=None*, *errors="strict"*)

Convert an OBMol into a structure record in the given format as a byte string

If *id* is not None then use it instead of the molecule's own title. Warning: this may briefly modify the molecule, so may not be thread-safe.

Parameters

- **mol** (*an Open Babel molecule*) – the molecule to use for the output
- **format** (*a format name string, or Format object*) – the output structure format
- **id** (*a string, or None to use the molecule's own id*) – an alternate record id
- **writer_args** (*a dictionary*) – writer arguments passed to the underlying toolkit
- **errors** (*one of "strict", "report", or "ignore"*) – specify how to handle errors

Returns a byte string

open_molecule_writer (openbabel_toolkit)

chemfp.openbabel_toolkit.**open_molecule_writer**(*destination=None*, *format=None*, *writer_args=None*, *errors="strict"*, *location=None*, *encoding="utf8"*, *encoding_errors="strict"*)

Return a MoleculeWriter which can write Open Babel molecules to a destination.

A *chemfp.base_toolkit.MoleculeWriter* has the methods `write_molecule`, `write_molecules`, and `write_ids_and_molecules`, which are ways to write an OBMol molecule, an OBMol molecule iterator, or an (id, OBMol molecule) pair iterator to a file.

Molecules are written to *destination*. The output format can be a string like "sdf.gz" or "smi", a *chemfp.base_toolkit.Format*, or Format-like object with "name" and "compression" attributes, or None to auto-detect based on the *destination*. If auto-detection is not possible, the output will be written as uncompressed SMILES.

The *writer_args* dictionary parameters depend on the format. Every format supports an `options` entry, which is passed to Open Babel's `SetOptions()`. See the Open Babel documentation for details. Some formats supports additional parameters:

•SMILES

–delimiter - one of "tab", "space", "to-eol", the space or tab characters, or None

–isomeric - True to write isomeric SMILES, False or default is non-isomeric

–canonicalization - True, “default”, or None uses Open Babel’s own canonicalization algorithm; False or “none” to use no canonicalization; “universal” generates a universal SMILES; “anticanonical” generates a SMILES with randomly assigned atom classes; “inchified” uses InChI-fied SMILES

•InChI and InChIKey

–delimiter - one of “tab”, “space”, “to-eol”, the space or tab characters, or None

–include_id - True or default to include the id as the second column; False has no id column

•SDF

–always_v3000 - True to always write V3000 files; False or default to write V3000 files only if needed.

–include_atom_class - True to include atom class; False or default does not

–include_hcount - True to include hcount; False or default does not

The *errors* parameter specifies how to handle errors. “strict” raises an exception, “report” sends a message to stderr and goes to the next record, and “ignore” goes to the next record.

The *location* parameter takes a *chemfp.io.Location* instance. If None then a default Location will be created.

Parameters

- **destination** (*a filename, file object, or None to write to stdout*) – the structure destination
- **format** (*a format name string, or Format(-like) object, or None to auto-detect*) – the output structure format
- **writer_args** (*a dictionary*) – writer arguments passed to the underlying toolkit
- **errors** (*one of "strict", "report", or "ignore"*) – specify how to handle errors
- **location** (a *chemfp.io.Location* object, or None) – object used to track writer state information

Returns a *chemfp.base_toolkit.MoleculeWriter* expecting Open Babel molecules

open_molecule_writer_to_string (openbabel_toolkit)

chemfp.openbabel_toolkit.**open_molecule_writer_to_string**(*format*, *writer_args=None*, *errors=”strict”*, *location=None*)

Return a MoleculeStringWriter which can write Open Babel molecule records to a string.

See *chemfp.openbabel_toolkit.open_molecule_writer()* for full parameter details.

Use the writer’s *chemfp.base_toolkit.MoleculeStringWriter.getvalue()* to get the output as a Unicode string.

Parameters

- **format** (*a format name string, or Format(-like) object, or None to auto-detect*) – the output structure format
- **writer_args** (*a dictionary*) – writer arguments passed to the underlying toolkit
- **errors** (*one of "strict", "report", or "ignore"*) – specify how to handle errors
- **location** (a *chemfp.io.Location* object, or None) – object used to track writer state information

Returns a *chemfp.base_toolkit.MoleculeStringWriter* expecting Open Babel molecules

open_molecule_writer_to_bytes (openbabel_toolkit)

chemfp.openbabel_toolkit.**open_molecule_writer_to_bytes**(*format*, *writer_args=None*, *errors="strict"*, *location=None*)

Return a MoleculeStringWriter which can write Open Babel molecule records to a byte string

See *chemfp.openbabel_toolkit.open_molecule_writer()* for full parameter details.

Use the writer's *chemfp.base_toolkit.MoleculeStringWriter.getvalue()* to get the output as a byte string.

Parameters

- **format** (*a format name string, or Format(-like) object, or None to auto-detect*) – the output structure format
- **writer_args** (*a dictionary*) – writer arguments passed to the underlying toolkit
- **errors** (*one of "strict", "report", or "ignore"*) – specify how to handle errors
- **location** (a *chemfp.io.Location* object, or None) – object used to track writer state information

Returns a *chemfp.base_toolkit.MoleculeStringWriter* expecting Open Babel molecules

copy_molecule (openbabel_toolkit)

chemfp.openbabel_toolkit.**copy_molecule**(*mol*)

Return a new OBMol molecule which is a copy of the given Open Babel molecule

Parameters **mol** (*an Open Babel molecule*) – the molecule to copy

Returns a new OBMol instance

add_tag (openbabel_toolkit)

chemfp.openbabel_toolkit.**add_tag**(*mol*, *tag*, *value*)

Add an SD tag value to the Open Babel molecule

Raises a KeyError if the tag is a special internal Open Babel name.

Parameters

- **mol** (*an Open Babel molecule*) – the molecule
- **tag** (*string*) – the SD tag name
- **value** (*string*) – the text for the tag

Returns None

get_tag (openbabel_toolkit)

chemfp.openbabel_toolkit.**get_tag**(*mol*, *tag*)
Get the named SD tag value, or None if it doesn't exist

Parameters

- **mol** (*an Open Babel molecule*) – the molecule
- **tag** (*string*) – the SD tag name

Returns a string, or None

get_tag_pairs (openbabel_toolkit)

chemfp.openbabel_toolkit.**get_tag_pairs**(*mol*)
Get a list of all SD tag (name, value) pairs for the molecule

Parameters **mol** (*an Open Babel molecule*) – the molecule

Returns a list of (string name, string value) pairs

get_id (openbabel_toolkit)

chemfp.openbabel_toolkit.**get_id**(*mol*)
Get the molecule's id using Open Babel's GetTitle()

Parameters **mol** (*an Open Babel molecule*) – the molecule

Returns a string

set_id (openbabel_toolkit)

chemfp.openbabel_toolkit.**set_id**(*mol*, *id*)
Set the molecule's id using Open Babel's SetTitle()

Parameters

- **mol** (*an Open Babel molecule*) – the molecule
- **id** (*string*) – the new id

Returns None

chemfp.openeye_toolkit module

The chemfp toolkit layer for OpenEye.

name

chemfp.openeye_toolkit.**name**

The string "openeye".

software

chemfp.openeye_toolkit.**software**

A string like "OEChem/20170208", where the second part of the string comes from OEChemGetVersion().

is_licensed (openeye_toolkit)

chemfp.openeye_toolkit.**is_licensed**()

Return True if the OEChem toolkit license is valid, otherwise False.

This does not check if the OEGraphSim license is valid. I haven't yet figured out how I want to handle that distinction. In the meanwhile you'll need to use the OEChem API yourself.

Returns True or False

get_formats (openeye_toolkit)

chemfp.openeye_toolkit.**get_formats**(*include_unavailable=False*)

Get the list of structure formats that OEChem supports

If *include_unavailable* is True then also include OEChem formats which aren't available to this specific version of OEChem.

Parameters **include_unavailable** (*True or False*) – include unavailable formats?

Returns a list of *chemfp.base_toolkit.Format* objects

get_input_formats (openeye_toolkit)

chemfp.openeye_toolkit.**get_input_formats**()

Get the list of supported OEChem input formats

Returns a list of *chemfp.base_toolkit.Format* objects

get_output_formats (openeye_toolkit)

chemfp.openeye_toolkit.**get_output_formats**()

Get the list of supported OEChem output formats

Returns a list of *chemfp.base_toolkit.Format* objects

get_format (openeye_toolkit)

chemfp.openeye_toolkit.**get_format**(*format*)
Get the named format, or raise a ValueError

This will raise a ValueError if OEChem does not implement the format *format_name* or that format is not available.

Parameters **format_name** (*a string*) – the format name

Returns a *chemfp.base_toolkit.Format* object

get_input_format (openeye_toolkit)

chemfp.openeye_toolkit.**get_input_format**(*format*)
Get the named input format, or raise a ValueError

This will raise a ValueError if OEChem does not implement the format *format_name* or that format is not an input format.

Parameters **format_name** (*a string*) – the format name

Returns a *chemfp.base_toolkit.Format* object

get_output_format (openeye_toolkit)

chemfp.openeye_toolkit.**get_output_format**(*format*)
Get the named format, or raise a ValueError

This will raise a ValueError if OEChem does not implement the format *format_name* or that format is not an output format.

Parameters **format_name** (*a string*) – the format name

Returns a *chemfp.base_toolkit.Format* object

get_input_format_from_source (openeye_toolkit)

chemfp.openeye_toolkit.**get_input_format_from_source**(*source=None*, *format=None*)
Get the most appropriate format given the available source and format information

If *format* is a *chemfp.base_toolkit.Format* then return it. If it's a Format-like object with "name" and "compression" attributes use it to make a real Format object with the same attributes. If it's a string then use it to create a Format object.

If *format* is None, use the *source* to auto-detect the format. If auto-detection is not possible, assume it's an uncompressed SMILES file.

Parameters

- **source** (*a filename (as a string), a file object, or None to read from stdin*) – the structure data source.
- **format** (*a Format(-like) object, string, or None*) – format information, if known.

Returns a *chemfp.base_toolkit.Format* object

get_output_format_from_destination (openeye_toolkit)

chemfp.openeye_toolkit.**get_output_format_from_destination**(*destination=None*, *format=None*)

Get the most appropriate format given the available destination and format information

If *format* is a *chemfp.base_toolkit.Format* then return it. If it's a Format-like object with "name" and "compression" attributes use it to make a real Format object with the same attributes. If it's a string then use it to create a Format object.

If *format* is None, use the *destination* to auto-detect the format. If auto-detection is not possible, assume it's an uncompressed SMILES file.

Parameters

- **destination** (*a filename (as a string), a file object, or None to read from stdin*) – the structure data source.
- **format** (*a Format(-like) object, string, or None*) – format information, if known.

Returns a *chemfp.base_toolkit.Format* object

read_molecules (openeye_toolkit)

chemfp.openeye_toolkit.**read_molecules**(*source=None*, *format=None*, *id_tag=None*, *reader_args=None*, *errors="strict"*, *location=None*, *encoding="utf8"*, *encoding_errors="strict"*)

Return an iterator that reads OEGraphMol molecules from a structure file

Iterate through the *format* structure records in *source*. If *format* is None then auto-detect the format based on the *source*. For SD files, use *id_tag* to get the record id from the given SD tag instead of the title line. (read_molecules() will ignore the *id_tag*. It exists to make it easier to switch between reader functions.)

Note: the reader will clear and reuse the OEGraphMol instance. Make a copy if you want to keep the molecule around.

The *reader_args* dictionary parameters depend on the format. Every OEChem format supports:

•aromaticity - one of "default", "openeye", "daylight", "tripos", "mdl", "mmff", or None

•flavor - a number, string-encoded number, or flavor string

A "flavor string" is a "|" or "," separated list of format-specific flavor terms. It can be a simple as "Default", or a more complex string like "Default|-ENDM|DELPHI" which for the PDB reader starts with the default settings, removes the ENDM flavor, and adds the CHARGE and RADIUS flavors.

The supported input flavor terms for each format are:

•SMILES - Canon, Strict, Default

•sdf - Default

•skc - Default

•mol2, mol2h - M2H, Default

•mmod - FormalCrg, Default

•pdb - ALL, ALTLOC, BondOrder, CHARGE, Connect, DATA, DELPHI, END, ENDM, FORMALCHARGE, FormalCrg, ImplicitH, RADIUS, Rings, SecStruct, TER, TerMask, Default

•xyz - BondOrder, Connect, FormalCrg, ImplicitH, Rings, Default

•cdx - SuperAtoms, Default

•oeb - Default

You can also pass in a numeric value like 123 or a numeric string like "0".

In addition, the SMILES record readers have limited support for the "delimiter" reader_arg:

•delimiter - one of "tab", "space", "to-eol", the space or tab characters, or None

Note: the first whitespace after the SMILES string will always be treated as a delimiter.

The *errors* parameter specifies how to handle errors. "strict" raises an exception, "report" sends a message to stderr and goes to the next record, and "ignore" goes to the next record.

The *location* parameter takes a *chemfp.io.Location* instance. If None then a default Location will be created.

See *chemfp.openeye_toolkit.read_ids_and_molecules()* if you want (id, OEGraphMol) pairs instead of just the molecules.

Parameters

- **source** (*a filename, file object, or None to read from stdin*) – the structure source
- **format** (*a format name string, or Format object, or None to auto-detect*) – the input structure format
- **id_tag** (*string, or None to use the record title*) – SD tag containing the record id
- **reader_args** (*a dictionary*) – reader parameters passed to the underlying toolkit
- **errors** (*one of "strict", "report", or "ignore"*) – specify how to handle errors
- **location** (a *chemfp.io.Location* object, or None) – object used to track parser state information

Returns a *chemfp.base_toolkit.MoleculeReader* iterating OEGraphMol molecules

read_molecules_from_string (openeye_toolkit)

chemfp.openeye_toolkit.**read_molecules_from_string**(*content*, *format*, *id_tag=None*, *reader_args=None*, *errors="strict"*, *location=None*)

Return an iterator that reads molecules from a string containing structure records

content is a string containing 0 or more records in the format *format*. See *chemfp.openeye_toolkit.read_molecules()* for details about the other parameters. See *chemfp.openeye_toolkit.read_ids_and_molecules_from_string()* if you want to read (id, OEGraphMol) pairs instead of just molecules.

Note: the reader will clear and reuse the OEGraphMol instance. Make a copy if you want to keep the molecule around.

Parameters

- **content** (*a string*) – the string containing structure records
- **format** (*a format name string, or Format object*) – the input structure format
- **id_tag** (*string, or None to use the record title*) – SD tag containing the record id
- **reader_args** (*a dictionary*) – reader arguments passed to the underlying toolkit
- **errors** (*one of "strict", "report", or "ignore"*) – specify how to handle errors
- **location** (a *chemfp.io.Location* object, or None) – object used to track parser state information

Returns a *chemfp.base_toolkit.MoleculeReader* iterating OEGraphMol molecules

read_ids_and_molecules (openeye_toolkit)

chemfp.openeye_toolkit.**read_ids_and_molecules**(*source=None*, *format=None*, *id_tag=None*, *reader_args=None*, *errors="strict"*, *location=None*, *encoding="utf8"*, *encoding_errors="strict"*)

Return an iterator that reads (id, OEGraphMol molecule) pairs from a structure file

See *chemfp.openeye_toolkit.read_molecules()* for full parameter details. The major difference is that this returns an iterator of (id, OEGraphMol) pairs instead of just the molecules.

Note: the reader will clear and reuse the OEGraphMol instance. Make a copy if you want to keep the molecule around.

Parameters

- **source** (*a filename, file object, or None to read from stdin*) – the structure source
- **format** (*a format name string, or Format object, or None to auto-detect*) – the input structure format
- **id_tag** (*string, or None to use the record title*) – SD tag containing the record id
- **reader_args** (*a dictionary*) – reader arguments passed to the underlying toolkit
- **errors** (*one of "strict", "report", or "ignore"*) – specify how to handle errors
- **location** (a *chemfp.io.Location* object, or None) – object used to track parser state information

Returns a *chemfp.base_toolkit.IdAndMoleculeReader* iterating (id, OEGraphMol) pairs

read_ids_and_molecules_from_string (openeye_toolkit)

chemfp.openeye_toolkit.**read_ids_and_molecules_from_string**(*content*, *format*, *id_tag=None*, *reader_args=None*, *errors="strict"*, *location=None*)

Return an iterator that reads (id, OEGraphMol) pairs from a string containing structure records

content is a string containing 0 or more records in the format *format*. See *chemfp.openeye_toolkit.read_molecules()* for details about the other parameters. See *chemfp.openeye_toolkit.read_molecules_from_string()* if you just want to read the OEGraphMol molecules instead of (id, OEGraphMol) pairs.

Note: the reader will clear and reuse the OEGraphMol instance. Make a copy if you want to keep the molecule around.

Parameters

- **content** (*a string*) – the string containing structure records
- **format** (*a format name string, or Format object*) – the input structure format
- **id_tag** (*string, or None to use the record title*) – SD tag containing the record id
- **reader_args** (*a dictionary*) – reader arguments passed to the underlying toolkit
- **errors** (*one of "strict", "report", or "ignore"*) – specify how to handle errors
- **location** (a *chemfp.io.Location* object, or None) – object used to track parser state information

Returns a *chemfp.base_toolkit.IdAndMoleculeReader* iterating (id, OEGraphMol) pairs

make_id_and_molecule_parser (openeye_toolkit)

chemfp.openeye_toolkit.**make_id_and_molecule_parser**(*format*, *id_tag=None*, *reader_args=None*, *errors="strict"*)

Create a specialized function which takes a record and returns an (id, OEGraphMol) pair

The returned function is optimized for reading many records from individual strings because it only does parameter validation once. The function will reuse the OEGraphMol for successive calls, so make a copy if you want to keep it around. However, I haven't really noticed much of a performance difference between this and *chemfp.openeye_toolkit.parse_id_and_molecule()* so I suggest you use that function directly instead of making a specialized function. (Let me know if making a specialized function is useful.)

See *chemfp.openeye_toolkit.read_molecules()* for details about the other parameters.

Parameters

- **format** (*a format name string, or Format object*) – the input structure format
- **id_tag** (*string, or None to use the record title*) – SD tag containing the record id
- **reader_args** (*a dictionary*) – reader arguments passed to the underlying toolkit
- **errors** (*one of "strict", "report", or "ignore"*) – specify how to handle errors

Returns a function of the form `parser(record string) -> (id, OEGraphMol)`

parse_molecule (openeye_toolkit)

chemfp.openeye_toolkit.**parse_molecule**(*content*, *format*, *id_tag=None*, *reader_args=None*, *errors="strict"*)

Parse the first structure record from the *content* string and return an OEGraphMol molecule.

content is a string containing a single structure record in format *format*. (Additional records are ignored). See *chemfp.openeye_toolkit.read_molecules()* for details about the other parameters. See *chemfp.openeye_toolkit.parse_id_and_molecule()* if you want the (id, OEGraphMol) pair instead of just the molecule.

Parameters

- **content** (*a string*) – the string containing a structure record
- **format** (*a format name string, or Format object*) – the input structure format
- **id_tag** (*string, or None to use the record title*) – SD tag containing the record id
- **reader_args** (*a dictionary*) – reader arguments passed to the underlying toolkit
- **errors** (*one of "strict", "report", or "ignore"*) – specify how to handle errors

Returns an OEGraphMol molecule

parse_id_and_molecule (openeye_toolkit)

chemfp.openeye_toolkit.**parse_id_and_molecule**(*content*, *format*, *id_tag=None*, *reader_args=None*, *errors="strict"*)

Parse the first structure record from *content* and return the (id, OEGraphMol) pair.

content is a string containing a single structure record in format *format*. (Additional records are ignored). See *chemfp.openeye_toolkit.read_molecules()* for details about the other parameters.

See *chemfp.openeye_toolkit.read_molecules()* for details about the other parameters. See *chemfp.openeye_toolkit.parse_molecule()* if just want the OEGraphMol molecule and not the the (id, OEGraphMol) pair.

Parameters

- **content** (*a string*) – the string containing a structure record
- **format** (*a format name string, or Format object*) – the input structure format
- **id_tag** (*string, or None to use the record title*) – SD tag containing the record id
- **reader_args** (*a dictionary*) – reader arguments passed to the underlying toolkit
- **errors** (*one of "strict", "report", or "ignore"*) – specify how to handle errors

Returns an (id, OEGraphMol molecule) pair

create_string (openeye_toolkit)

chemfp.openeye_toolkit.**create_string**(*mol*, *format*, *id=None*, *writer_args=None*, *errors="strict"*)

Convert an OEChem molecule into a structure record in the given format as a Unicode string

If *id* is not None then use it instead of the molecule's own title. Warning: this may briefly modify the molecule, so may not be thread-safe.

Parameters

- **mol** (*an OEChem molecule*) – the molecule to use for the output
- **format** (*a format name string, or Format object*) – the output structure format
- **id** (*a string, or None to use the molecule's own id*) – an alternate record id
- **writer_args** (*a dictionary*) – writer arguments passed to the underlying toolkit
- **errors** (*one of "strict", "report", or "ignore"*) – specify how to handle errors

Returns a string

create_bytes (openeye_toolkit)

chemfp.openeye_toolkit.**create_bytes**(*mol*, *format*, *id=None*, *writer_args=None*, *errors="strict"*)

Convert an OEChem molecule into a structure record in the given format as a byte string

If *id* is not None then use it instead of the molecule's own title. Warning: this may briefly modify the molecule, so may not be thread-safe.

Parameters

- **mol** (*an OEChem molecule*) – the molecule to use for the output

- **format** (*a format name string, or Format object*) – the output structure format
- **id** (*a string, or None to use the molecule's own id*) – an alternate record id
- **writer_args** (*a dictionary*) – writer arguments passed to the underlying toolkit
- **errors** (*one of "strict", "report", or "ignore"*) – specify how to handle errors

Returns a string

open_molecule_writer (openeye_toolkit)

chemfp.openeye_toolkit.**open_molecule_writer**(*destination=None*, *format=None*, *writer_args=None*, *errors="strict"*, *location=None*, *encoding="utf8"*, *encoding_errors="strict"*)

Return a MoleculeWriter which can write OEChem molecules to a destination.

A *chemfp.base_toolkit.MoleculeWriter* has the methods `write_molecule`, `write_molecules`, and `write_ids_and_molecules`, which are ways to write an OEChem molecule, an OEChem molecule iterator, or an (id, OEChem molecule) pair iterator to a file.

Molecules are written to *destination*. The output format can be a string like "sdf.gz" or "smi", a *chemfp.base_toolkit.Format*, or Format-like object with "name" and "compression" attributes, or None to auto-detect based on the *destination*. If auto-detection is not possible, the output will be written as uncompressed SMILES.

The *writer_args* dictionary parameters depend on the format. Every OEChem format supports:

•aromaticity - one of "default", "openeye", "daylight", "tripos", "mdl", "mmff", or None

•flavor - a number, string-encoded number, or flavor string

A "flavor string" is a "|" or "," separated list of format-specific flavor terms. It can be as simple as "Default", or a more complex string like DEFAULT|-AtomStereo|-BondSterol|Canonical to generate a canonical SMILES string without stereo information.

The supported output flavor terms for each format are:

•SMILES - AtomMaps, AtomStereo, BondStereo, Canonical, ExtBonds, Hydrogens, ImpHCount, Isotopes, Kekule, RGroups, SuperAtoms

•sdf - CurrentParity, MCHG, MDLParity, MISO, MRGP, MV30, NoParity, Default

•mol2, mol2h - AtomNames, AtomTypeNames, BondTypeNames, Hydrogens, OrderAtoms, Substructure, Default

•sln - Default

•pdb - BONDS, BOTH, CHARGE, CurrentResidues, DELPHI, ELEMENT, FORMALCHARGE, FormalCrg, HETBONDS, NoResidues, OEResidues, ORDERS, OrderAtoms, RADIUS, TER, Default

•xyz - Charges, Symbols, Default

•cdx - Default

•mopac - CHARGES, XYZ, Default

- mf - Title, Default
- oeb - Default
- inchi, inchikey - Chiral, FixedHLayer, Hydrogens, ReconnectedMetals, Stereo, RelativeStereo, RacemicStereo, Default

You can also pass in a numeric value like 123 or a numeric string like "0".

The *errors* parameter specifies how to handle errors. "strict" raises an exception, "report" sends a message to stderr and goes to the next record, and "ignore" goes to the next record.

The *location* parameter takes a *chemfp.io.Location* instance. If None then a default Location will be created.

Parameters

- **destination** (*a filename, file object, or None to write to stdout*) – the structure destination
- **format** (*a format name string, or Format(-like) object, or None to auto-detect*) – the output structure format
- **writer_args** (*a dictionary*) – writer parameters passed to the underlying toolkit
- **errors** (*one of "strict", "report", or "ignore"*) – specify how to handle errors
- **location** (a *chemfp.io.Location* object, or None) – object used to track writer state information

Returns a *chemfp.base_toolkit.MoleculeWriter* expecting OEChem molecules

open_molecule_writer_to_string (openeye_toolkit)

chemfp.openeye_toolkit.**open_molecule_writer_to_string**(*format*, *writer_args=None*, *errors="strict"*, *location=None*)

Return a MoleculeStringWriter which can write OEChem molecule records to a Unicode string.

See *chemfp.openeye_toolkit.open_molecule_writer()* for full parameter details.

Use the writer's *chemfp.base_toolkit.MoleculeStringWriter.getvalue()* to get the output string as a Unicode string.

Parameters

- **format** (*a format name string, or Format(-like) object, or None to auto-detect*) – the output structure format
- **writer_args** (*a dictionary*) – writer arguments passed to the underlying toolkit
- **errors** (*one of "strict", "report", or "ignore"*) – specify how to handle errors
- **location** (a *chemfp.io.Location* object, or None) – object used to track writer state information

Returns a *chemfp.base_toolkit.MoleculeStringWriter* expecting OEChem molecules

open_molecule_writer_to_bytes (openeye_toolkit)

chemfp.openeye_toolkit.**open_molecule_writer_to_bytes**(*format*, *writer_args=None*, *errors="strict"*, *location=None*)

Return a MoleculeStringWriter which can write OEChem molecule records to a byte string.

See *chemfp.openeye_toolkit.open_molecule_writer()* for full parameter details.

Use the writer's *chemfp.base_toolkit.MoleculeStringWriter.getvalue()* to get the output string as a byte string.

Parameters

- **format** (*a format name string, or Format(-like) object, or None to auto-detect*) – the output structure format
- **writer_args** (*a dictionary*) – writer arguments passed to the underlying toolkit
- **errors** (*one of "strict", "report", or "ignore"*) – specify how to handle errors
- **location** (a *chemfp.io.Location* object, or None) – object used to track writer state information

Returns a *chemfp.base_toolkit.MoleculeStringWriter* expecting OEChem molecules

copy_molecule (openeye_toolkit)

chemfp.openeye_toolkit.**copy_molecule**(*mol*)

Return a new OEGraphMol which is a copy of the given OEChem molecule

Parameters **mol** (*an Open Babel molecule*) – the molecule to copy

Returns a new OBMol instance

add_tag (openeye_toolkit)

chemfp.openeye_toolkit.**add_tag**(*mol*, *tag*, *value*)

Add an SD tag value to the OEChem molecule

Parameters

- **mol** (*an OEChem molecule*) – the molecule
- **tag** (*string*) – the SD tag name
- **value** (*string*) – the text for the tag

Returns None

get_tag (openeye_toolkit)

chemfp.openeye_toolkit.**get_tag**(*mol*, *tag*)
Get the named SD tag value, or None if it doesn't exist

Parameters

- **mol** (*an OEChem molecule*) – the molecule
- **tag** (*string*) – the SD tag name

Returns a string, or None

get_tag_pairs (openeye_toolkit)

chemfp.openeye_toolkit.**get_tag_pairs**(*mol*)
Get a list of all SD tag (name, value) pairs for the molecule

Parameters **mol** (*an OEChem molecule*) – the molecule

Returns a list of (string name, string value) pairs

get_id (openeye_toolkit)

chemfp.openeye_toolkit.**get_id**(*mol*)
Get the molecule's id using OEChem's GetTitle()

Parameters **mol** (*an OEChem molecule*) – the molecule

Returns a string

set_id (openeye_toolkit)

chemfp.openeye_toolkit.**set_id**(*mol*, *id*)
Set the molecule's id using OEChem's SetTitle()

Parameters

- **mol** (*an OEChem molecule*) – the molecule
- **id** (*string*) – the new id

Returns None

chemfp.rdkit_toolkit module

The chemfp toolkit layer for RDKit.

name

chemfp.rdkit_toolkit.**name**

The string "rdkit".

software

chemfp.rdkit_toolkit.**software**

A string like "RDKit/2016.09.3", where the second part of the string comes from rdkit.rdBase.rdkitVersion.

is_licensed (rdkit_toolkit)

chemfp.rdkit_toolkit.**is_licensed**()

Return True - RDKit is always licensed

Returns True

get_formats (rdkit_toolkit)

chemfp.rdkit_toolkit.**get_formats**(*include_unavailable=False*)

Get the list of structure formats that RDKit supports

If *include_unavailable* is True then also include RDKit formats which aren't available to this specific version of RDKit, such as the InChI formats if your RDKit installation wasn't compiled with InChI support.

Parameters **include_unavailable** (*True or False*) – include unavailable formats?

Returns a list of Format objects

get_input_formats (rdkit_toolkit)

chemfp.rdkit_toolkit.**get_input_formats**()

Get the list of supported RDKit input formats

Returns a list of *chemfp.base_toolkit.Format* objects

get_output_formats (rdkit_toolkit)

chemfp.rdkit_toolkit.**get_output_formats**()

Get the list of supported RDKit output formats

Returns a list of *chemfp.base_toolkit.Format* objects

get_format (rdkit_toolkit)

chemfp.rdkit_toolkit.**get_format**(*format*)

Get the named format, or raise a ValueError

This will raise a ValueError if RDKit does not implement the format *format_name* or that format is not available.

Parameters **format_name** (*a string*) – the format name

Returns a list of *chemfp.base_toolkit.Format* objects

get_input_format (rdkit_toolkit)

chemfp.rdkit_toolkit.**get_input_format**(*format*)

Get the named input format, or raise a ValueError

This will raise a ValueError if RDKit does not implement the format *format_name* or that format is not an input format.

Parameters **format_name** (*a string*) – the format name

Returns a list of *chemfp.base_toolkit.Format* objects

get_output_format (rdkit_toolkit)

chemfp.rdkit_toolkit.**get_output_format**(*format*)

Get the named format, or raise a ValueError

This will raise a ValueError if RDKit does not implement the format *format_name* or that format is not an output format.

Parameters **format_name** (*a string*) – the format name

Returns a list of *chemfp.base_toolkit.Format* objects

get_input_format_from_source (rdkit_toolkit)

chemfp.rdkit_toolkit.**get_input_format_from_source**(*source=None*, *format=None*)

Get the most appropriate format given the available source and format information

If *format* is a *chemfp.base_toolkit.Format* then return it. If it's a Format-like object with "name" and "compression" attributes use it to make a real Format object with the same attributes. If it's a string then use it to create a Format object.

If *format* is None, use the *source* to auto-detect the format. If auto-detection is not possible, assume it's an uncompressed SMILES file.

Parameters

- **source** (*a filename (as a string), a file object, or None to read from stdin*) – the structure data source.
- **format** (*a Format(-like) object, string, or None*) – format information, if known.

Returns a *chemfp.base_toolkit.Format* object

get_output_format_from_destination (rdkit_toolkit)

chemfp.rdkit_toolkit.**get_output_format_from_destination**(*destination=None*, *format=None*)

Get the most appropriate format given the available destination and format information

If *format* is a *chemfp.base_toolkit.Format* then return it. If it's a Format-like object with "name" and "compression" attributes use it to make a real Format object with the same attributes. If it's a string then use it to create a Format object.

If *format* is None, use the *destination* to auto-detect the format. If auto-detection is not possible, assume it's an uncompressed SMILES file.

Parameters

- **destination** (*a filename (as a string), a file object, or None to read from stdin*) – The structure data source.
- **format** (*a Format(-like) object, string, or None*) – format information, if known.

Returns a *chemfp.base_toolkit.Format* object

read_molecules (rdkit_toolkit)

chemfp.rdkit_toolkit.**read_molecules**(*source=None*, *format=None*, *id_tag=None*, *reader_args=None*, *errors="strict"*, *location=None*, *encoding="utf8"*, *encoding_errors="strict"*)

Return an iterator that reads RDKit molecules from a structure file

Iterate through the *format* structure records in *source*. If *format* is None then auto-detect the format based on the *source*. For SD files, use *id_tag* to get the record id from the given SD tag instead of the title line. (read_molecules() will ignore the *id_tag*. It exists to make it easier to switch between reader functions.)

Note: the reader returns a new RDKit molecule each time.

The *reader_args* dictionary parameters depend on the format. These include:

•SMILES

–delimiter - one of "tab", "space", "to-eol", the space or tab characters, or None

–has_header - True or False

–sanitize - True or default sanitizes; False for unsanitized processing

•InChI

–delimiter - one of "tab", "space", "to-eol", the space or tab characters, or None

–sanitize - True or default sanitizes; False for unsanitized processing

–removeHs - True or default removes explicit hydrogens; False leaves them in the structure

–logLevel - an integer log level

–treatWarningAsError - True raises an exception on error; False or default keeps processing

•SDF

–sanitize - True or default sanitizes; False for unsanitized processing

–removeHs - True or default removes explicit hydrogens; False leaves them in the structure

–strictParsing - True or default for strict parsing; False for lenient parsing

The *errors* parameter specifies how to handle errors. "strict" raises an exception, "report" sends a message to stderr and goes to the next record, and "ignore" goes to the next record.

The *location* parameter takes a *chemfp.io.Location* instance. If None then a default Location will be created.

See *chemfp.rdkit_toolkit.read_ids_and_molecules()* if you want (id, molecule) pairs instead of just the molecules.

Parameters

- **source** (*a filename, file object, or None to read from stdin*) – the structure source
- **format** (*a format name string, or Format object, or None to auto-detect*) – the input structure format
- **id_tag** (*string, or None to use the record title*) – SD tag containing the record id
- **reader_args** (*a dictionary*) – reader parameters passed to the underlying toolkit
- **errors** (*one of "strict", "report", or "ignore"*) – specify how to handle errors
- **location** (a *chemfp.io.Location* object, or None) – object used to track parser state information

Returns a *chemfp.base_toolkit.MoleculeReader* iterating RDKit molecules

read_molecules_from_string (rdkit_toolkit)

chemfp.rdkit_toolkit.**read_molecules_from_string**(*content*, *format*, *id_tag=None*, *reader_args=None*, *errors="strict"*, *location=None*)

Return an iterator that reads RDKit molecules from a string containing structure records

content is a string containing 0 or more records in the format *format*. See *chemfp.rdkit_toolkit.read_molecules()* for details about the other parameters. See *chemfp.rdkit_toolkit.read_ids_and_molecules_from_string()* if you want to read (id, RDKit) pairs instead of just molecules.

Parameters

- **content** (*a string*) – the string containing structure records
- **format** (*a format name string, or Format object*) – the input structure format
- **id_tag** (*string, or None to use the record title*) – SD tag containing the record id
- **reader_args** (*a dictionary*) – reader arguments passed to the underlying toolkit
- **errors** (*one of "strict", "report", or "ignore"*) – specify how to handle errors
- **location** (a *chemfp.io.Location* object, or None) – object used to track parser state information

Returns a *chemfp.base_toolkit.MoleculeReader* iterating RDKit molecules

read_ids_and_molecules (rdkit_toolkit)

chemfp.rdkit_toolkit.**read_ids_and_molecules**(*source=None*, *format=None*, *id_tag=None*, *reader_args=None*, *errors="strict"*, *location=None*, *encoding="utf8"*, *encoding_errors="strict"*)

Return an iterator that reads (id, RDKit molecule) pairs from a structure file

See *chemfp.rdkit_toolkit.read_molecules()* for full parameter details. The major difference is that this returns an iterator of (id, RDKit molecule) pairs instead of just the molecules.

Parameters

- **source** (*a filename, file object, or None to read from stdin*) – the structure source
- **format** (*a format name string, or Format object, or None to auto-detect*) – the input structure format
- **id_tag** (*string, or None to use the record title*) – SD tag containing the record id
- **reader_args** (*a dictionary*) – reader arguments passed to the underlying toolkit
- **errors** (*one of "strict", "report", or "ignore"*) – specify how to handle errors
- **location** (a *chemfp.io.Location* object, or None) – object used to track parser state information

Returns a *chemfp.base_toolkit.IdAndMoleculeReader* iterating (id, RDKit molecule) pairs

read_ids_and_molecules_from_string (rdkit_toolkit)

chemfp.rdkit_toolkit.**read_ids_and_molecules_from_string**(*content*, *format*, *id_tag=None*, *reader_args=None*, *errors="strict"*, *location=None*)

Return an iterator that reads (id, RDKit molecule) pairs from a string containing structure records

content is a string containing 0 or more records in the format *format*. See *chemfp.rdkit_toolkit.read_molecules()* for details about the other parameters. See *chemfp.rdkit_toolkit.read_molecules_from_string()* if you just want to read the RDKit molecules instead of (id, molecule) pairs.

Parameters

- **content** (*a string*) – the string containing structure records
- **format** (*a format name string, or Format object*) – the input structure format

- **id_tag** (*string, or None to use the record title*) – SD tag containing the record id
- **reader_args** (*a dictionary*) – reader arguments passed to the underlying toolkit
- **errors** (*one of "strict", "report", or "ignore"*) – specify how to handle errors
- **location** (a *chemfp.io.Location* object, or None) – object used to track parser state information

Returns a *chemfp.base_toolkit.IdAndMoleculeReader* iterating (id, RDKit molecule) pairs

make_id_and_molecule_parser (rdkit_toolkit)

chemfp.rdkit_toolkit.**make_id_and_molecule_parser**(*format*, *id_tag=None*, *reader_args=None*, *errors="strict"*)

Create a specialized function which takes a record and returns an (id, RDKit molecule) pair

The returned function is optimized for reading many records from individual strings because it only does parameter validation once. However, I haven't really noticed much of a performance difference between this and *chemfp.rdkit_toolkit.parse_id_and_molecule()* so you can probably so I suggest you use that function directly instead of making a specialized function. (Let me know if making a specialized function is useful.)

See *chemfp.rdkit_toolkit.read_molecules()* for details about the other parameters.

Parameters

- **format** (*a format name string, or Format object*) – the input structure format
- **id_tag** (*string, or None to use the record title*) – SD tag containing the record id
- **reader_args** (*a dictionary*) – reader arguments passed to the underlying toolkit
- **errors** (*one of "strict", "report", or "ignore"*) – specify how to handle errors

Returns a function of the form `parser(record string) -> (id, RDKit molecule)`

parse_molecule (rdkit_toolkit)

chemfp.rdkit_toolkit.**parse_molecule**(*content*, *format*, *id_tag=None*, *reader_args=None*, *errors="strict"*)

Parse the first structure record from the *content* string and return an RDKit molecule.

content is a string containing a single structure record in format *format*. (Additional records are ignored). See *chemfp.rdkit_toolkit.read_molecules()* for details about the other parameters. See *chemfp.rdkit_toolkit.parse_id_and_molecule()* if you want the (id, RDKit molecule) pair instead of just the molecule.

Parameters

- **content** (*a string*) – the string containing a structure record
- **format** (*a format name string, or Format object*) – the input structure format
- **id_tag** (*string, or None to use the record title*) – SD tag containing the record id
- **reader_args** (*a dictionary*) – reader arguments passed to the underlying toolkit
- **errors** (*one of "strict", "report", or "ignore"*) – specify how to handle errors

Returns an RDKit molecule

parse_id_and_molecule (rdkit_toolkit)

chemfp.rdkit_toolkit.**parse_id_and_molecule**(*content*, *format*, *id_tag=None*, *reader_args=None*, *errors="strict"*)

Parse the first structure record from *content* and return the (id, RDKit molecule) pair.

content is a string containing a single structure record in format *format*. (Additional records are ignored). See *chemfp.rdkit_toolkit.read_molecules()* for details about the other parameters.

See *chemfp.rdkit_toolkit.read_molecules()* for details about the other parameters. See *chemfp.rdkit_toolkit.parse_molecule()* if just want the RDKit molecule and not the the (id, RDKit molecule) pair.

Parameters

- **content** (*a string*) – the string containing a structure record
- **format** (*a format name string, or Format object*) – the input structure format
- **id_tag** (*string, or None to use the record title*) – SD tag containing the record id
- **reader_args** (*a dictionary*) – reader arguments passed to the underlying toolkit
- **errors** (*one of "strict", "report", or "ignore"*) – specify how to handle errors

Returns an (id, RDKit molecule) pair

create_string (rdkit_toolkit)

chemfp.rdkit_toolkit.**create_string**(*mol*, *format*, *id=None*, *writer_args=None*, *errors="strict"*)

Convert an RDKit molecule into a structure record in the given format as a Unicode string

If *id* is not None then use it instead of the molecule's own title. Warning: this may briefly modify the molecule, so may not be thread-safe.

Parameters

- **mol** (*an RDKit molecule*) – the molecule to use for the output

- **format** (*a format name string, or Format object*) – the output structure format
- **id** (*a string, or None to use the molecule's own id*) – an alternate record id
- **writer_args** (*a dictionary*) – writer arguments passed to the underlying toolkit
- **errors** (*one of "strict", "report", or "ignore"*) – specify how to handle errors

Returns a Unicode string

create_bytes (rdkit_toolkit)

chemfp.rdkit_toolkit.**create_bytes**(*mol*, *format*, *id=None*, *writer_args=None*, *errors="strict"*)

Convert an RDKit molecule into a structure record in the given format as a byte string

If *id* is not None then use it instead of the molecule's own title. Warning: this may briefly modify the molecule, so may not be thread-safe.

Parameters

- **mol** (*an RDKit molecule*) – the molecule to use for the output
- **format** (*a format name string, or Format object*) – the output structure format
- **id** (*a string, or None to use the molecule's own id*) – an alternate record id
- **writer_args** (*a dictionary*) – writer arguments passed to the underlying toolkit
- **errors** (*one of "strict", "report", or "ignore"*) – specify how to handle errors

Returns a byte string

open_molecule_writer (rdkit_toolkit)

chemfp.rdkit_toolkit.**open_molecule_writer**(*destination=None*, *format=None*, *writer_args=None*, *errors="strict"*, *location=None*, *encoding="utf8"*, *encoding_errors="strict"*)

Return a MoleculeWriter which can write RDKit molecules to a destination.

A *chemfp.base_toolkit.MoleculeWriter* has the methods `write_molecule`, `write_molecules`, and `write_ids_and_molecules`, which are ways to write an RDKit molecule, an RDKit molecule iterator, or an (id, RDKit molecule) pair iterator to a file.

Molecules are written to *destination*. The output format can be a string like "sdf.gz" or "smi", a *chemfp.base_toolkit.Format*, or Format-like object with "name" and "compression" attributes, or None to auto-detect based on the *destination*. If auto-detection is not possible, the output will be written as uncompressed SMILES.

The *writer_args* dictionary parameters depend on the format. These include:

•SMILES

–delimiter - one of “tab”, “space”, “to-eol”, the space or tab characters, or None

–isomericSmiles - True to generate isomeric SMILES

–kekuleSmiles - True to generate SMILES in Kekule form

–canonical - True to generate a canonical SMILES

–allBondsExplicit - True to write explict ‘-‘ and ‘:’ bonds, even if they can be inferred; default is False

InChI and InChIKey

•delimiter - one of “tab”, “space”, “to-eol”, the space or tab characters, or None

•include_id - True or default to include the id as the second column; False has no id column

•options - an options string passed to the underlying InChI library

•logLevel - an integer log level

•treatWarningAsError - True raises an exception on error; False or default keeps processing

SDF

•includeStereo - True include stereo information; False or default does not

•kekulize - True or default creates the connection table with bonds in Kekeule form

The *errors* parameter specifies how to handle errors. “strict” raises an exception, “report” sends a message to stderr and goes to the next record, and “ignore” goes to the next record.

The *location* parameter takes a *chemfp.io.Location* instance. If None then a default Location will be created.

Parameters

- **destination** (*a filename, file object, or None to write to stdout*) – the structure destination
- **format** (*a format name string, or Format(-like) object, or None to auto-detect*) – the output structure format
- **writer_args** (*a dictionary*) – writer parameters passed to the underlying toolkit
- **errors** (*one of "strict", "report", or "ignore"*) – specify how to handle errors
- **location** (a *chemfp.io.Location* object, or None) – object used to track writer state information

Returns a *chemfp.base_toolkit.MoleculeWriter* expecting RDKit molecules

open_molecule_writer_to_string (rdkit_toolkit)

chemfp.rdkit_toolkit.**open_molecule_writer_to_string**(*format*, *writer_args=None*, *errors="strict"*, *location=None*)

Return a MoleculeStringWriter which can write molecule records in the given format to a string.

See *chemfp.rdkit_toolkit.open_molecule_writer()* for full parameter details.

Use the writer's *chemfp.base_toolkit.MoleculeStringWriter.getvalue()* to get the output as a Unicode string.

Parameters

- **format** (*a format name string, or Format(-like) object, or None to auto-detect*) – the output structure format
- **writer_args** (*a dictionary*) – writer arguments passed to the underlying toolkit
- **errors** (*one of "strict", "report", or "ignore"*) – specify how to handle errors
- **location** (a *chemfp.io.Location* object, or None) – object used to track writer state information

Returns a *chemfp.base_toolkit.MoleculeStringWriter* expecting RDKit molecules

open_molecule_writer_to_bytes (rdkit_toolkit)

chemfp.rdkit_toolkit.**open_molecule_writer_to_bytes**(*format*, *writer_args=None*, *errors="strict"*, *location=None*)

Return a MoleculeStringWriter which can write molecule records in the given format to a text string.

See *chemfp.rdkit_toolkit.open_molecule_writer()* for full parameter details.

Use the writer's *chemfp.base_toolkit.MoleculeStringWriter.getvalue()* to get the output as a byte string.

Parameters

- **format** (*a format name string, or Format(-like) object, or None to auto-detect*) – the output structure format
- **writer_args** (*a dictionary*) – writer arguments passed to the underlying toolkit
- **errors** (*one of "strict", "report", or "ignore"*) – specify how to handle errors
- **location** (a *chemfp.io.Location* object, or None) – object used to track writer state information

Returns a *chemfp.base_toolkit.MoleculeStringWriter* expecting RDKit molecules

copy_molecule (rdkit_toolkit)

chemfp.rdkit_toolkit.**copy_molecule**(*mol*)

Return a new RDKit molecule which is a copy of the given molecule

Parameters **mol** (*an RDKit molecule*) – the molecule to copy

Returns a new RDKit Mol instance

add_tag (rdkit_toolkit)

chemfp.rdkit_toolkit.**add_tag**(*mol*, *tag*, *value*)
Add an SD tag value to the RDKit molecule

Parameters

- **mol** (*an RDKit molecule*) – the molecule
- **tag** (*string*) – the SD tag name
- **value** (*string*) – the text for the tag

Returns None

get_tag (rdkit_toolkit)

chemfp.rdkit_toolkit.**get_tag**(*mol*, *tag*)
Get the named SD tag value, or None if it doesn't exist

Parameters

- **mol** (*an RDKit molecule*) – the molecule
- **tag** (*string*) – the SD tag name

Returns a string, or None

get_tag_pairs (rdkit_toolkit)

chemfp.rdkit_toolkit.**get_tag_pairs**(*mol*)
Get a list of all SD tag (name, value) pairs for the molecule

Parameters **mol** (*an RDKit molecule*) – the molecule

Returns a list of (string name, string value) pairs

get_id (rdkit_toolkit)

chemfp.rdkit_toolkit.**get_id**(*mol*)
Get the molecule's id from RDKit's _Name property

Parameters **mol** (*an RDKit molecule*) – the molecule

Returns a string

set_id (rdkit_toolkit)

chemfp.rdkit_toolkit.**set_id**(*mol*, *id*)
Set the molecule's id as RDKit's _Name property

Parameters

- **mol** (*an RDKit molecule*) – the molecule
- **id** (*string*) – the new id

Returns None

chemfp.text_toolkit module

The text_toolkit implements the chemfp toolkit API but where the "molecules" are simple TextRecord instances which store the records as text strings. It does not use a back-end chemistry toolkit, and it cannot convert between different chemistry representations.

The TextRecord is a base class. The actual records depend on the format, and will be one of:

- *SDFRecord*
- *SmiRecord*
- *CanRecord*
- *UsmRecord*
- *SmiStringRecord*
- *CanStringRecord*
- *UsmStringRecord*

The text toolkit will let you "convert" between the different SMILES formats, but it doesn't actually change the SMILES string. The SMILES records have the attributes `id`, `record` and `smiles`.

The toolkit also knows a bit about the SD format. The SDF records have the attributes `id`, `id_bytes` and `record`, and there are methods to get SD tag values and add a tag to the end of the tag data block.

The text_toolkit also supports a few SDF-specific I/O functions to read SDF records directly as a string instead of wrapped in a TextRecord.

The record types also have the attributes `encoding` and `encoding_errors` which affect how the record bytes are parsed.

name

`chemfp.text_toolkit.`**`name`**

The string "text"

software

`chemfp.text_toolkit.`**`software`**

A string like "chemfp/3.0".

is_licensed (text_toolkit)

`chemfp.text_toolkit.`**`is_licensed`**`()`

Return True - chemfp's text toolkit is always licensed

Returns True

get_formats (text_toolkit)

chemfp.text_toolkit.**get_formats**(*include_unavailable=False*)
Get the list of structure formats that chemfp's text toolkit supports

This version of chemfp will always support the structure formats available to chemfp so 'include_unavailable' does not affect anything. (It may affect other toolkits.)

Parameters **include_unavailable** – include unavailable formats?

Value include_unavailable True or False

Returns a list of *chemfp.base_toolkit.Format* objects

get_input_formats (text_toolkit)

chemfp.text_toolkit.**get_input_formats**()
Get the list of supported chemfp text toolkit input formats

Returns a list of *chemfp.base_toolkit.Format* objects

get_output_formats (text_toolkit)

chemfp.text_toolkit.**get_output_formats**()
Get the list of supported chemfp text toolkit output formats

Returns a list of *chemfp.base_toolkit.Format* objects

get_format (text_toolkit)

chemfp.text_toolkit.**get_format**(*format_name*)
Get the named format, or raise a ValueError

This will raise a ValueError for unknown format names.

Parameters **format_name** – the format name

Value format_name a string

Returns a *chemfp.base_toolkit.Format* object

get_input_format (text_toolkit)

chemfp.text_toolkit.**get_input_format**(*format_name*)
Get the named input format, or raise a ValueError

This will raise a ValueError for unknown format names or if that format is not an input format.

Parameters **format_name** – the format name

Value format_name a string

Returns a *chemfp.base_toolkit.Format* object

get_output_format (text_toolkit)

chemfp.text_toolkit.**get_output_format**(*format_name*)

Get the named format, or raise a ValueError

This will raise a ValueError for unknown format names or if that format is not an output format.

Parameters **format_name** – the format name

Value format_name a string

Returns a *chemfp.base_toolkit.Format* object

get_input_format_from_source (text_toolkit)

chemfp.text_toolkit.**get_input_format_from_source**(*source=None*, *format=None*)

Get the most appropriate format given the available source and format information

If *format* is a *chemfp.base_toolkit.Format* then return it. If it's a Format-like object with "name" and "compression" attributes use it to make a real Format object with the same attributes. If it's a string then use it to create a Format object.

If *format* is None, use the *source* to auto-detect the format. If auto-detection is not possible, assume it's an uncompressed SMILES file.

Parameters

- **source** (*A filename (as a string), a file object, or None to read from stdin*) – The structure data source.
- **format** (*A Format(-like) object, string, or None*) – Format information, if known.

Returns a *chemfp.base_toolkit.Format* object

get_output_format_from_destination (text_toolkit)

chemfp.text_toolkit.**get_output_format_from_destination**(*destination=None*, *format=None*)

Get the most appropriate format given the available destination and format information

If *format* is a *chemfp.base_toolkit.Format* then return it. If it's a Format-like object with "name" and "compression" attributes use it to make a real Format object with the same attributes. If it's a string then use it to create a Format object.

If *format* is None, use the *destination* to auto-detect the format. If auto-detection is not possible, assume it's an uncompressed SMILES file.

Parameters

- **destination** (*A filename (as a string), a file object, or None to read from stdin*) – The structure data source.
- **format** (*A Format(-like) object, string, or None*) – format information, if known.

Returns A *chemfp.base_toolkit.Format* object

read_molecules (text_toolkit)

chemfp.text_toolkit.**read_molecules**(*source=None*, *format=None*, *id_tag=None*, *reader_args=None*, *errors="strict"*, *location=None*, *encoding="utf8"*, *encoding_errors="strict"*)

Return an iterator that reads TextRecord instances from a structure file

Iterate through the *format* structure records in *source*. If *format* is None then auto-detect the format based on the *source*. For SD files, use *id_tag* to get the record id from the given SD tag instead of the title line. (read_molecules() will ignore the *id_tag*. It exists to make it easier to switch between reader functions.)

Only the SMILES formats use the *reader_args* dictionary. The supported parameters are:

•delimiter - one of "tab", "space", "to-eol", the space or tab characters, or None

•has_header - True or False

The *errors* parameter specifies how to handle errors. "strict" raises an exception, "report" sends a message to stderr and goes to the next record, and "ignore" goes to the next record.

The *location* parameter takes a *chemfp.io.Location* instance. If None then a default Location will be created.

See *read_ids_and_molecules()* if you want (id, *TextRecord*) pairs instead of just the molecules.

Parameters

- **source** (*a filename, file object, or None to read from stdin*) – the structure source
- **format** (*a format name string, or Format object, or None to auto-detect*) – the input structure format
- **id_tag** (*string, or None to use the record title*) – SD tag containing the record id
- **reader_args** (*a dictionary*) – reader parameters passed to the underlying toolkit
- **errors** (*one of "strict", "report", or "ignore"*) – specify how to handle errors
- **location** (a *chemfp.io.Location* object, or None) – object used to track parser state information
- **encoding** (*string (typically 'utf8' or 'latin1')*) – the byte encoding
- **encoding_errors** (*string (typically 'strict', 'ignore', or 'replace')*) – how to handle decoding failure

Returns a *chemfp.base_toolkit.MoleculeReader* iterating *TextRecord* molecules

read_molecules_from_string (text_toolkit)

chemfp.text_toolkit.**read_molecules_from_string**(*content*, *format*, *id_tag=None*, *reader_args=None*, *errors="strict"*, *location=None*)

Return an iterator that reads TextRecord instances from a string containing structure records

content is a string containing 0 or more records in the format *format*. See `read_molecules()` for details about the other parameters. See `read_ids_and_molecules_from_string()` if you want to read (id, `TextRecord`) pairs instead of just molecules.

Parameters

- **content** (*a string*) – the string containing structure records
- **format** (*a format name string, or Format object*) – the input structure format
- **id_tag** (*string, or None to use the record title*) – SD tag containing the record id
- **reader_args** (*a dictionary*) – reader arguments passed to the underlying toolkit
- **errors** (*one of "strict", "report", or "ignore"*) – specify how to handle errors
- **location** (a `chemfp.io.Location` object, or None) – object used to track parser state information
- **encoding** (*string (typically 'utf8' or 'latin1')*) – the byte encoding
- **encoding_errors** (*string (typically 'strict', 'ignore', or 'replace')*) – how to handle decoding failure

Returns a `chemfp.base_toolkit.MoleculeReader` iterating `TextRecord` molecules

read_ids_and_molecules (text_toolkit)

chemfp.text_toolkit.**read_ids_and_molecules**(*source=None*, *format=None*, *id_tag=None*, *reader_args=None*, *errors="strict"*, *location=None*, *encoding="utf8"*, *encoding_errors="strict"*)

Return an iterator that reads (id, TextRecord) pairs from a structure file

See `chemfp.text_toolkit.read_molecules()` for full parameter details. The major difference is that this returns an iterator of (id, `TextRecord`) pairs instead of just the molecules.

Parameters

- **source** (*a filename, file object, or None to read from stdin*) – the structure source
- **format** (*a format name string, or Format object, or None to auto-detect*) – the input structure format

- **id_tag** (*string, or None to use the record title*) – SD tag containing the record id
- **reader_args** (*a dictionary*) – reader arguments passed to the underlying toolkit
- **errors** (*one of "strict", "report", or "ignore"*) – specify how to handle errors
- **location** (a *chemfp.io.Location* object, or None) – object used to track parser state information
- **encoding** (*string (typically 'utf8' or 'latin1')*) – the byte encoding
- **encoding_errors** (*string (typically 'strict', 'ignore', or 'replace')*) – how to handle decoding failure

Returns a chemfp.text_toolkit.IdAndMoleculeReader iterating (id, *TextRecord*) pairs

read_ids_and_molecules_from_string (text_toolkit)

chemfp.text_toolkit.**read_ids_and_molecules_from_string**(*content*, *format*, *id_tag=None*, *reader_args=None*, *errors="strict"*, *location=None*)

Return an iterator that reads (id, TextRecord) pairs from a string containing structure records

content is a string containing 0 or more records in the format *format*. See *chemfp.rdkit_toolkit.read_molecules()* for details about the other parameters. See *chemfp.rdkit_toolkit.read_molecules_from_string()* if you just want to read the *TextRecord* molecules instead of (id, TextRecord) pairs.

Parameters

- **content** (*a string*) – the string containing structure records
- **format** (*a format name string, or Format object*) – the input structure format
- **id_tag** (*string, or None to use the record title*) – SD tag containing the record id
- **reader_args** (*a dictionary*) – reader arguments passed to the underlying toolkit
- **errors** (*one of "strict", "report", or "ignore"*) – specify how to handle errors
- **location** (a *chemfp.io.Location* object, or None) – object used to track parser state information
- **encoding** (*string (typically 'utf8' or 'latin1')*) – the byte encoding
- **encoding_errors** (*string (typically 'strict', 'ignore', or 'replace')*) – how to handle decoding failure

Returns a *chemfp.base_toolkit.IdAndMoleculeReader* iterating (id, *TextRecord*) pairs

make_id_and_molecule_parser (text_toolkit)

chemfp.text_toolkit.**make_id_and_molecule_parser**(*format*, *id_tag=None*, *reader_args=None*, *errors="strict"*)

Create a specialized function which takes a record and returns an (id, TextRecord) pair

The returned function is optimized for reading many records from individual strings because it only does parameter validation once. However, I haven't really noticed much of a performance difference between this and *chemfp.text_toolkit.parse_id_and_molecule()* so I suggest you use that function directly instead of making a specialized function. (Let me know if making a specialized function is useful.)

See *chemfp.text_toolkit.read_molecules()* for details about the other parameters. The specific *TextRecord* subclass returned depends on the format.

Parameters

- **format** (*a format name string, or Format object*) – the input structure format
- **id_tag** (*string, or None to use the record title*) – SD tag containing the record id
- **reader_args** (*a dictionary*) – reader arguments passed to the underlying toolkit
- **errors** (*one of "strict", "report", or "ignore"*) – specify how to handle errors

Returns a function of the form `parser(record string) -> (id, text_record)`

parse_molecule (text_toolkit)

chemfp.text_toolkit.**parse_molecule**(*content*, *format*, *id_tag=None*, *reader_args=None*, *errors="strict"*)

Parse the first structure record from the *content* string and return a TextRecord.

content is a string containing a single structure record in format *format*. (Additional records are ignored). See *chemfp.text_toolkit.read_molecules()* for details about the other parameters. See *chemfp.text_toolkit.parse_id_and_molecule()* if you want the (id, *TextRecord*) pair instead of just the text record.

Parameters

- **content** (*a string*) – the string containing a structure record
- **format** (*a format name string, or Format object*) – the input structure format
- **id_tag** (*string, or None to use the record title*) – SD tag containing the record id
- **reader_args** (*a dictionary*) – reader arguments passed to the underlying toolkit
- **errors** (*one of "strict", "report", or "ignore"*) – specify how to handle errors

- **encoding** (*string (typically 'utf8' or 'latin1')*) – the byte encoding
- **encoding_errors** (*string (typically 'strict', 'ignore', or 'replace')*) – how to handle decoding failure

Returns a *TextRecord*

parse_id_and_molecule (text_toolkit)

chemfp.text_toolkit.**parse_id_and_molecule**(*content*, *format*, *id_tag=None*, *reader_args=None*, *errors="strict"*)

Parse the first structure record from *content* and return the (id, TextRecord) pair.

content is a string containing a single structure record in format *format*. (Additional records are ignored). See *chemfp.rdkit_toolkit.read_molecules()* for details about the other parameters.

See *chemfp.rdkit_toolkit.read_molecules()* for details about the other parameters. See *chemfp.rdkit_toolkit.parse_molecule()* if just want the *TextRecord* and not the the (id, TextRecord) pair.

Parameters

- **content** (*a string*) – the string containing a structure record
- **format** (*a format name string, or Format object*) – the input structure format
- **id_tag** (*string, or None to use the record title*) – SD tag containing the record id
- **reader_args** (*a dictionary*) – reader arguments passed to the underlying toolkit
- **errors** (*one of "strict", "report", or "ignore"*) – specify how to handle errors
- **encoding** (*string (typically 'utf8' or 'latin1')*) – the byte encoding
- **encoding_errors** (*string (typically 'strict', 'ignore', or 'replace')*) – how to handle decoding failure

Returns an (id, *TextRecord* molecule) pair

create_string (text_toolkit)

chemfp.text_toolkit.**create_string**(*mol*, *format*, *id=None*, *writer_args=None*, *errors="strict"*)

Convert a TextRecord into a structure record in the given format as a Unicode string

If *id* is not None then use it instead of the molecule's own id.

Parameters

- **mol** (a *TextRecord*) – the molecule to use for the output
- **format** (*a format name string, or Format object*) – the output structure format

- **id** (*a string, or None to use the molecule's own id*) – an alternate record id
- **writer_args** (*a dictionary*) – writer arguments passed to the underlying toolkit
- **errors** (*one of "strict", "report", or "ignore"*) – specify how to handle errors

Returns a Unicode string

create_bytes (text_toolkit)

chemfp.text_toolkit.**create_bytes**(*mol*, *format*, *id=None*, *writer_args=None*, *errors="strict"*)

Convert a TextRecord into a structure record in the given format as a byte string

If *id* is not None then use it instead of the molecule's own id.

Parameters

- **mol** (a *TextRecord*) – the molecule to use for the output
- **format** (*a format name string, or Format object*) – the output structure format
- **id** (*a string, or None to use the molecule's own id*) – an alternate record id
- **writer_args** (*a dictionary*) – writer arguments passed to the underlying toolkit
- **errors** (*one of "strict", "report", or "ignore"*) – specify how to handle errors

Returns a byte string

open_molecule_writer (text_toolkit)

chemfp.text_toolkit.**open_molecule_writer**(*destination=None*, *format=None*, *writer_args=None*, *errors="strict"*, *location=None*, *encoding="utf8"*, *encoding_errors="strict"*)

Return a MoleculeWriter which can write TextRecord instances to a destination.

A *chemfp.base_toolkit.MoleculeWriter* has the methods `write_molecule`, `write_molecules`, and `write_ids_and_molecules`, which are ways to write an *TextRecord*, an TextRecord iterator, or an (id, TextRecord) pair iterator to a file.

TextRecords are written to *destination*. The output format can be a string like "sdf.gz" or "smi", a *chemfp.base_toolkit.Format*, or Format-like object with "name" and "compression" attributes, or None to auto-detect based on the *destination*. If auto-detection is not possible, the output will be written as uncompressed SMILES.

That said, the text toolkit doesn't know how to convert between SMILES and SDF formats, and will raise an exception if you try.

The *writer_args* is only used for the "smi", "can", and "usm" output formats. The only supported parameter is:

```
* delimiter - one of "tab", "space", "to-eol", the space or tab␣
↪characters, or None
```

The *errors* parameter specifies how to handle errors. "strict" raises an exception, "report" sends a message to stderr and goes to the next record, and "ignore" goes to the next record.

The *location* parameter takes a *chemfp.io.Location* instance. If None then a default Location will be created.

Parameters

- **destination** (*a filename, file object, or None to write to stdout*) – the structure destination
- **format** (*a format name string, or Format(-like) object, or None to auto-detect*) – the output structure format
- **writer_args** (*a dictionary*) – writer arguments passed to the underlying toolkit
- **errors** (*one of "strict", "report", or "ignore"*) – specify how to handle errors
- **location** (a *chemfp.io.Location* object, or None) – object used to track writer state information
- **encoding** (*string (typically 'utf8' or 'latin1')*) – the byte encoding
- **encoding_errors** (*string (typically 'strict', 'ignore', or 'replace')*) – how to handle decoding failure

Returns a *chemfp.base_toolkit.MoleculeWriter* expecting *TextRecord* instances

open_molecule_writer_to_string (text_toolkit)

chemfp.text_toolkit.**open_molecule_writer_to_string**(*format*, *writer_args=None*, *errors="strict"*, *location=None*)

Return a MoleculeStringWriter which can write TextRecord instances to a string.

See *chemfp.text_toolkit.open_molecule_writer()* for full parameter details.

Use the writer's *chemfp.base_toolkit.MoleculeStringWriter.getvalue()* to get the output as a Unicode string.

Parameters

- **format** (*a format name string, or Format(-like) object, or None to auto-detect*) – the output structure format
- **writer_args** (*a dictionary*) – writer arguments passed to the underlying toolkit
- **errors** (*one of "strict", "report", or "ignore"*) – specify how to handle errors
- **location** (a *chemfp.io.Location* object, or None) – object used to track writer state information

Returns a *chemfp.base_toolkit.MoleculeStringWriter* expecting *TextRecord* instances

open_molecule_writer_to_bytes (text_toolkit)

chemfp.text_toolkit.**open_molecule_writer_to_bytes**(*format*, *writer_args=None*, *errors="strict"*, *location=None*)

Return a MoleculeStringWriter which can write TextRecord instances to a string.

See *chemfp.text_toolkit.open_molecule_writer()* for full parameter details.

Use the writer's *chemfp.base_toolkit.MoleculeStringWriter.getvalue()* to get the output as a byte string.

Parameters

- **format** (*a format name string, or Format(-like) object, or None to auto-detect*) – the output structure format
- **writer_args** (*a dictionary*) – writer arguments passed to the underlying toolkit
- **errors** (*one of "strict", "report", or "ignore"*) – specify how to handle errors
- **location** (a *chemfp.io.Location* object, or None) – object used to track writer state information

Returns a *chemfp.base_toolkit.MoleculeStringWriter* expecting *TextRecord* instances

copy_molecule (text_toolkit)

chemfp.text_toolkit.**copy_molecule**(*mol*)

Return a new TextRecord which is a copy of the given TextRecord

Parameters **mol** (a *TextRecord*) – the text record

Returns a new *TextRecord*

add_tag (text_toolkit)

chemfp.text_toolkit.**add_tag**(*mol*, *tag*, *value*)

Add an SD tag value to the TextRecord

If the *mol* is in "sdf" format then this will modify `mol.record` to append the new *tag* and *value* to the end of the tag block. The other tags will not be modified, including tags with the same tag name.

Parameters

- **mol** (a *TextRecord*) – the text record
- **tag** (*string*) – the SD tag name
- **value** (*string*) – the text for the tag

Returns None

get_tag (text_toolkit)

chemfp.text_toolkit.**get_tag**(*mol*, *tag*)
Get the named SD tag value, or None if it doesn't exist

If the *mol* is in "sdf" format then this will return the corresponding tag value from `mol.record`, or None if the tag does not exist.

If the record is in any other format then it will return None.

Parameters

- **mol** (a *TextRecord*) – the molecule
- **tag** (*string*) – the SD tag name

Returns a string, or None

get_tag_pairs (text_toolkit)

chemfp.text_toolkit.**get_tag_pairs**(*mol*)
Get a list of all SD tag (name, value) pairs for the TextRecord

If the *mol* is in "sdf" format then this will return the list of (tag, value) pairs in `mol.record`, where the *tag* and *value* are strings.

If the record is in any other format then it will return an empty list.

Parameters **mol** (a *TextRecord*) – the molecule

Returns a list of (tag name, tag value) pairs

get_id (text_toolkit)

chemfp.text_toolkit.**get_id**(*mol*)
Get the molecule's id from the TextRecord's id field

This is toolkit-portable way to get `mol.id`.

Parameters **mol** (*a TextRecord*) – the molecule

Returns a string

set_id (text_toolkit)

chemfp.text_toolkit.**set_id**(*mol*, *id*)
Set the TextRecord's id to the new id

This is the toolkit-portable way to write `mol.id = id`.

Note: this does not modify `mol.record`. Use *chemfp.text_toolkit.create_string()* or similar text_toolkit functions to get the record text with a new identifier.

Parameters

- **mol** (a *TextRecord*) – the molecule
- **id** (*string*) – the new id

Returns None

read_sdf_records (text_toolkit)

chemfp.text_toolkit.**read_sdf_records**(*source=None*, *reader_args=None*, *compression=None*, *errors="strict"*, *location=None*, *block_size=327680*)

Return an iterator that reads each record from an SD file as a string.

Iterate through the records in *source*, which must be in SD format. If *compression* is None or "auto" then auto-detect the compression type based on *source*, and default to uncompressed when it can't be determined. Use "gz" when the input is gzip compressed, and "none" or "" if uncompressed.

The *reader_args* parameter is currently unused. It exists for future compatability.

The *errors* parameter specifies how to handle errors. "strict" raises an exception, "report" sends a message to stderr and goes to the next record, and "ignore" goes to the next record.

The *location* parameter takes a *chemfp.io.Location* instance. If None then a default Location will be created.

The *block_size* parameter is the number of bytes to read from the SD file. The current implementation reads a block, iterates through the records in the block, then prepends any remaining text to the start of the next block. You shouldn't need to change this parameter, but if you do, please let me know.

Note: to prevent accidental memory consumption if the input is in the wrong format, a complete record must be found within the first 327680 bytes or 5*block_size bytes, whichever is larger.

The parser has only a basic understanding of the SD format. It knows how to handle the counts line, the SKP property, and even tag data with the value '$$$$'. It is not a full validator and it does not know chemistry.

WARNING: the parser does not yet handle the MS Windows newline convention.

See *read_sdf_ids_and_records()* if you want (id, record) pairs, and *read_sdf_ids_and_values()* if you want (id, tag data) pairs. See *read_sdf_ids_and_records_from_string()* to read from a string instead of a file or file-like object.

Parameters

- **source** (*a filename, file object, or None to read from stdin*) – the SDF source
- **reader_args** (*currently ignored*) – currently ignored
- **compression** (*one of "auto", "none", "", or "gz"*) – the data content compression method
- **errors** (*one of "strict", "report", or "ignore"*) – specify how to handle errors
- **location** (a *chemfp.io.Location* object, or None) – object used to track parser state information

Returns a *chemfp.base_toolkit.RecordReader()* iterating over the records as a string

read_sdf_ids_and_records (text_toolkit)

chemfp.text_toolkit.**read_sdf_ids_and_records**(*source=None*, *id_tag=None*, *reader_args=None*, *compression=None*, *errors="strict"*, *location=None*, *encoding="utf8"*, *encoding_errors="strict"*, *block_size=327680*)

Return an iterator that reads the (id, record string) pairs from an SD file

See *read_sdf_records()* for most parameter details. That function iterates over the records, while this one iterates over the (id, record) pairs. By default the id comes from the title line. Use *id_tag* to get the record id from the given SD tag instead.

See *read_sdf_ids_and_values()* if you want to read an identifier and tag value, or two tag values, instead of returning the full record.

Parameters

- **source** (*a filename, file object, or None to read from stdin*) – the SDF source
- **id_tag** (*string, or None to use the record title*) – SD tag containing the record id
- **reader_args** (*currently ignored*) – currently ignored
- **compression** (*one of "auto", "none", "", or "gz"*) – the data content compression method
- **errors** (*one of "strict", "report", or "ignore"*) – specify how to handle errors
- **location** (a *chemfp.io.Location* object, or None) – object used to track parser state information

Returns a *chemfp.base_toolkit.IdAndRecordReader* iterating (id, record string) pairs

read_sdf_ids_and_values (text_toolkit)

chemfp.text_toolkit.**read_sdf_ids_and_values**(*source=None*, *id_tag=None*, *value_tag=None*, *reader_args=None*, *compression=None*, *errors="strict"*, *location=None*, *encoding="utf8"*, *encoding_errors="strict"*, *block_size=327680*)

Return an iterator that reads the (id, tag value string) pairs from an SD file

See *read_sdf_records()* for most parameter details. That function iterates over the records, while this one iterates over the (id, tag value) pairs.

By default this uses the title line for both the id and tag value strings. Use *id_tag* and *value_tag*, respectively, to use a given tag value instead. If a tag doesn't exist then None will be used.

Parameters

- **source** (*a filename, file object, or None to read from stdin*) – the SDF source

- **id_tag** (*string, or None to use the record title*) – SD tag containing the record id
- **value_tag** (*string, or None to use the record title*) – SD tag containing the value
- **reader_args** (*currently ignored*) – currently ignored
- **compression** (*one of "auto", "none", "", or "gz"*) – the data content compression method
- **errors** (*one of "strict", "report", or "ignore"*) – specify how to handle errors
- **location** (a *chemfp.io.Location* object, or None) – object used to track parser state information

Returns a *chemfp.base_toolkit.IdAndRecordReader* iterating (id, value string) pairs

read_sdf_records_from_string (text_toolkit)

chemfp.text_toolkit.**read_sdf_records_from_string**(*content*, *reader_args=None*, *compression=None*, *errors="strict"*, *location=None*, *block_size=327680*)

Return an iterator that reads each record from a string containing SD records

See *read_sdf_records_from_string()* for the parameter details. The main difference is that this function reads from *content*, which is a string containing 0 or more SDF records.

If content is a (Unicode) string then it must only contain ASCII characters, the records will be returned as strings, and the compression option is not supported. If content is a byte string then the records will be returned as byte strings, and compression is supported.

See *read_sdf_ids_and_records_from_string()* to read (id, record) pairs and *read_sdf_ids_and_values_from_string()* to read (id, tag value) pairs.

Parameters

- **content** (*string or bytes*) – a string containing zero or more SD records
- **reader_args** (*currently ignored*) – currently ignored
- **compression** (*one of "auto", "none", "", or "gz"*) – the data content compression method
- **errors** (*one of "strict", "report", or "ignore"*) – specify how to handle errors
- **location** (a *chemfp.io.Location* object, or None) – object used to track parser state information

Returns a *chemfp.base_toolkit.RecordReader* iterating over each record as a string

read_sdf_ids_and_records_from_string (text_toolkit)

chemfp.text_toolkit.**read_sdf_ids_and_records_from_string**(*content=None*, *id_tag=None*, *reader_args=None*, *compression=None*, *errors="strict"*, *location=None*, *encoding="utf8"*, *encoding_errors="strict"*, *block_size=327680*)

Return an iterator that reads the (id, record) pairs from a string containing SD records

This function reads the records from *content*, which is a string containing 0 or more SDF records. It iterates over the (id, record) pairs. By default the id comes from the first line of the SD record. Use *id_tag* to use a given tag value instead. See *read_sdf_records()* for details about the other parameters.

If content is a (Unicode) string then it must only contain ASCII characters, the records will be returned as strings, the compression option is not supported, and the encoding and encoding_errors parameters are ignored.

If content is a byte string then the records will be returned as byte strings, compression is supported, and the encoding and encoding_errors parameters are used to parse the id.

Parameters

- **content** (*string or bytes*) – a string containing zero or more SD records
- **id_tag** (*string, or None to use the record title*) – SD tag containing the record id
- **reader_args** (*currently ignored*) – currently ignored
- **compression** (*one of "auto", "none", "", or "gz"*) – the data content compression method
- **errors** (*one of "strict", "report", or "ignore"*) – specify how to handle errors
- **location** (a *chemfp.io.Location* object, or None) – object used to track parser state information

Returns a *chemfp.base_toolkit.IdAndRecordReader* iterating over the (id, record string) pairs

read_sdf_ids_and_values_from_string (text_toolkit)

chemfp.text_toolkit.**read_sdf_ids_and_values_from_string**(*content=None*, *id_tag=None*, *value_tag=None*, *compression=None*, *reader_args=None*, *errors="strict"*, *location=None*, *encoding="utf8"*, *encoding_errors="strict"*, *block_size=327680*)

Return an iterator that reads the (id, value) pairs from a string containing SD records

This function reads the records from *content*, which is a string containing 0 or more SDF records. It iterates over the (id, value) pairs, which by default both contain the title line. Use *id_tag* and *value_tag*, respectively, to use a given tag value instead. If a tag doesn't exist then None will be used.

If content is a (Unicode) string then it must only contain ASCII characters, the compression option is not supported, and the encoding and encoding_errors parameters are ignored.

If content is a byte string then the records will be returned as byte strings, compression is supported, and the encoding and encoding_errors parameters are used to parse the id and value.

See *read_sdf_records()* for details about the other parameters.

Parameters

- **content** (*string or bytes*) – a string containing zero or more SD records
- **id_tag** (*string, or None to use the record title*) – SD tag containing the record id
- **value_tag** (*string, or None to use the record title*) – SD tag containing the value
- **reader_args** (*currently ignored*) – currently ignored
- **compression** (*one of "auto", "none", "", or "gz"*) – the data content compression method
- **errors** (*one of "strict", "report", or "ignore"*) – specify how to handle errors
- **location** (a *chemfp.io.Location* object, or None) – object used to track parser state information

Returns a *chemfp.base_toolkit.IdAndRecordReader* iterating over the (id, value) pairs

get_sdf_tag (text_toolkit)

chemfp.text_toolkit.**get_sdf_tag**(*sdf_record*, *tag*)

Return the value for a named tag in an SDF record string

Get the value for the tag named *tag* from the string *sdf_record* containing an SD record.

Parameters

- **sdf_record** (*string*) – an SD record
- **tag** (*string*) – a tag name

Returns the corresponding tag value as a string, or None

add_sdf_tag (text_toolkit)

chemfp.text_toolkit.**add_sdf_tag**(*sdf_record*, *tag*, *value*)

Add an SD tag value to an SD record string

This will append the new *tag* and *value* to the end of the tag data block in the *sdf_record* string.

Parameters

- **sdf_record** (*string*) – an SD record
- **tag** (*string*) – a tag name
- **value** (*string*) – the new tag value

Returns a new SD record string with the new tag and value

get_sdf_tag_pairs (text_toolkit)

chemfp.text_toolkit.**get_sdf_tag_pairs**(*sdf_record*)

Return the (tag, value) entries in the SDF record string

Parse the *sdf_record* and return the tag data as a list of (tag, value) pairs. The type of the returned strings will be the same as the type of the input sdf_record string.

Parameters **sdf_record** (*string*) – an SDF record

Returns a list of (tag, value) pairs

get_sdf_id (text_toolkit)

chemfp.text_toolkit.**get_sdf_id**(*sdf_record*)

Return the id for the SDF record string

The id is the first line of the *sdf_record*. A future version of this function may support an *id_tag* parameter. Let me know if that would be useful.

The returned id string will have the same type as the input sdf_record.

Parameters **sdf_record** (*string*) – an SD record

Returns the first line of the SD record

set_sdf_id (text_toolkit)

chemfp.text_toolkit.**set_sdf_id**(*sdf_record*, *id*)

Set the id of the SDF record string to a new value

Set the first line of *sdf_record* to the new *id*, which must not contain a newline.

The sdf_record and the id must have the same string type.

Parameters

- **sdf_record** (*string*) – an SDF record
- **id** (*string*) – the new id

chemfp._text_toolkit module (private)

As you might have infered from the leading "_" in "_text_toolkit", this is not a public module. There is no reason for you to import it directly, the module name is subject to change, and even the location of the classes is also subject to change. The reason why I even bring it up is because the *chemfp.text_toolkit* returns class instances from this module, so you might well wonder about them.

TextRecord

class chemfp._text_toolkit.**TextRecord**

Base class for the text_toolkit 'molecules', which work with the records as text.

The *chemfp.text_toolkit* implements the toolkit API, but it doesn't know chemistry. Instead of returning real molecule objects, with atoms and bonds, it returns TextRecord subclass instances that hold the record as a text string.

As an implementation detail (which means its subject to change) there is a subclass for each of the support formats.

- *SDFRecord* - holds "sdf" records
- *SmiRecord* - holds "smi" records (the full line from a "smi" SMILES file)
- *CanRecord* - holds "can" records (the full line from a "can" SMILES file)
- *UsmRecord* - holds "usm" records (the full line from a "usm" SMILES file)
- *SmiStringRecord* - holds "smistring" records (only the "smistring" SMILES string; no id)
- *CanStringRecord* - holds "canstring" records (only the "canstring" SMILES string; no id)
- *UsmStringRecord* - holds "usmstring" records (only the "usmstring" SMILES string; no id)

All of the classes have the following attributes: .. py:attribute:: id

The record identifier as a Unicode string, or None if there is no identifier

id_bytes
The record identifier as a byte string, or None if there is no identifier

record
The record, as a string. For the smistring, canstring, and usmstring formats, this is only the SMILES string.

record_format
One of "sdf", "smi", "can", "usm", "smistring", "canstring", or "usmstring".

The SMILES classes have an attribute:

smiles
The SMILES string component of the record.

add_tag(*tag*, *value*)
Add an SD tag value to the TextRecord

This methods does nothing if the record is not an "sdf" record.

Parameters

- **tag** (*string*) – the SD tag name
- **value** (*string*) – the text for the tag

Returns None

get_tag(*tag*)
Get the named SD tag value, or None if it doesn't exist or is not an "sdf" record.

Parameters **tag** (*byte or Unicode string*) – the SD tag name

Returns a Unicode string, or None

get_tag_as_bytes(*tag*)
Get the named SD tag value, or None if it doesn't exist or is not an "sdf" record.

Parameters **tag** (*byte string*) – the SD tag name

Returns a byte string, or None

get_tag_pairs()
Get a list of all SD tag (name, value) pairs for the TextRecord using Unicode strings

This function returns an empty list if the record is not an "sdf" record.

Returns a list of (Unicode string name, Unicode string value) pairs

get_tag_pairs_as_bytes()
Get a list of all SD tag (name, value) pairs for the TextRecord using byte strings

This function returns an empty list if the record is not an "sdf" record.

Returns a list of (byte string name, byte string value) pairs

copy()
Return a new record which is a copy of the given record

SDFRecord

class chemfp._text_toolkit.**SDFRecord**
Holds an SDF record. See *chemfp._text_toolkit.TextRecord* for API details

SmiRecord

class chemfp._text_toolkit.**SmiRecord**
Holds an "smi" record. See *chemfp._text_toolkit.TextRecord* for API details

CanRecord

class chemfp._text_toolkit.**CanRecord**
Holds an "can" record. See *chemfp._text_toolkit.TextRecord* for API details

UsmRecord

class chemfp._text_toolkit.**UsmRecord**
Holds an "usm" record. See *chemfp._text_toolkit.TextRecord* for API details

SmiStringRecord

class chemfp._text_toolkit.**SmiStringRecord**
Holds an "smistring" record. See *chemfp._text_toolkit.TextRecord* for API details

CanStringRecord

class chemfp._text_toolkit.**CanStringRecord**
Holds an "canstring" record. See *chemfp._text_toolkit.TextRecord* for API details

UsmStringRecord

class chemfp._text_toolkit.**UsmStringRecord**
Holds an "usmstring" record. See *chemfp._text_toolkit.TextRecord* for API details

chemfp.io module

This module implements a single public class, *Location*, which tracks parser state information, including the location of the current record in the file. The other functions and classes are undocumented, should not be used, and may change in future releases.

Location

class chemfp.io.**Location**

Get location and other internal reader and writer state information

A Location instance gives a way to access information like the current record number, line number, and molecule object.:

```
>>> import chemfp
>>> with chemfp.read_molecule_fingerprints("RDKit-MACCS166",
...                         "ChEBI_lite.sdf.gz", id_tag="ChEBI ID") as␣
↪reader:
...   for id, fp in reader:
...     if id == "CHEBI:3499":
...         print("Record starts at line", reader.location.lineno)
...         print("Record byte range:", reader.location.offsets)
...         print("Number of atoms:", reader.location.mol.GetNumAtoms())
...         break
...
[08:18:12]  S group MUL ignored on line 103
Record starts at line 3599
Record byte range: (138171, 141791)
Number of atoms: 36
```

The supported properties are:

- filename - a string describing the source or destination
- lineno - the line number for the start of the file
- mol - the toolkit molecule for the current record
- offsets - the (start, end) byte positions for the current record
- output_recno - the number of records written successfully
- recno - the current record number
- record - the record as a text string
- record_format - the record format, like "sdf" or "can"

Most of the readers and writers do not support all of the properties. Unsupported properties return a None. The *filename* is a read/write attribute and the other attributes are read-only.

If you don't pass a location to the readers and writers then they will create a new one based on the source or destination, respectively. You can also pass in your own Location, created as `Location(filename)` if you have an actual filename, or `Location.from_source(source)` or `Location.from_destination(destination)` if you have a more generic source or destination.

__init__ (*filename=None*)
Use *filename* as the location's filename

from_source (*cls*, *source*)
Create a Location instance based on the source

If *source* is a string then it's used as the filename. If *source* is None then the location filename is "<stdin>". If *source* is a file object then its `name` attribute is used as the filename, or None if there is no attribute.

from_destination (*cls*, *destination*)
Create a Location instance based on the destination

If *destination* is a string then it's used as the filename. If *destination* is None then the location filename is "<stdout>". If *destination* is a file object then its `name` attribute is used as the filename, or None if there is no attribute.

__repr__ ()
Return a string like 'Location("<stdout>")'

first_line
Read-only attribute.

The first line of the current record

filename
Read/write attribute.

A string which describes the source or destination. This is usually the source or destination filename but can be a string like "<stdin>" or "<stdout>".

mol
Read-only attribute.

The molecule object for the current record

offsets
Read-only attribute.

The (start, end) byte offsets, starting from 0

start is the record start byte position and *end* is one byte past the last byte of the record.

output_recno
Read-only attribute.

The number of records actually written to the file or string.

The value `recno - output_recno` is the number of records sent to the writer but which had an error and could not be written to the output.

recno
Read-only attribute.

The current record number

For writers this is the number of records sent to the writer, and output_recno is the number of records sucessfully written to the file or string.

record
Read-only attribute.

The current record as an uncompressed text string

record_format
Read-only attribute.

The record format name

where()
Return a human readable description about the current reader or writer state.

The description will contain the filename, line number, record number, and up to the first 40 characters of the first line of the record, if those properties are available.

CHAPTER 2

License and advertisement

This program was developed by Andrew Dalke <dalke@dalkescientific.com>, Andrew Dalke Scientific, AB. It is distributed under the "MIT" license, shown below.

Further chemfp development depends on funding from people like you. Asking for voluntary contributions almost never works. Instead, starting with chemfp 1.1, the source code is distributed under an incentive program. You can pay for the commerical distribution, or use the no-cost version.

If you pay for the commercial distribution then you will get the most recent version of chemfp, free upgrades for one year, support, and a discount on renewing participation in the incentive program.

I also maintain the chemfp-1.x series. Version chemfp-1.3 is available at no cost from chemfp.com, or if you know someone with chemfp 2.x or 3.x you might be able to get it from them at no cost. It's free/open source software, after all.

If you have questions about or with to purchase the commercial distribution, send an email to sales@dalkescientific.com .

```
Copyright (c) 2010-2017 Andrew Dalke Scientific, AB (Sweden)

Permission is hereby granted, free of charge, to any person obtaining
a copy of this software and associated documentation files (the
"Software"), to deal in the Software without restriction, including
without limitation the rights to use, copy, modify, merge, publish,
distribute, sublicense, and/or sell copies of the Software, and to
permit persons to whom the Software is furnished to do so, subject to
the following conditions:

The above copyright notice and this permission notice shall be
included in all copies or substantial portions of the Software.

THE SOFTWARE IS PROVIDED "AS IS", WITHOUT WARRANTY OF ANY KIND,
EXPRESS OR IMPLIED, INCLUDING BUT NOT LIMITED TO THE WARRANTIES OF
MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE AND
NONINFRINGEMENT. IN NO EVENT SHALL THE AUTHORS OR COPYRIGHT HOLDERS BE
LIABLE FOR ANY CLAIM, DAMAGES OR OTHER LIABILITY, WHETHER IN AN ACTION
```

```
OF CONTRACT, TORT OR OTHERWISE, ARISING FROM, OUT OF OR IN CONNECTION
WITH THE SOFTWARE OR THE USE OR OTHER DEALINGS IN THE SOFTWARE.
```

Copyright to portions of the code are held by other people or organizations, and may be under a different license. See the specific code for details. These are:

- OpenMP, cpuid, POPCNT, and Lauradoux implementations by Kim Walisch, <kim.walisch@gmail.com>, under the MIT license
- SSSE3.2 popcount implementation by Stanford Univeristy (written by Imran S. Haque <ihaque@cs.stanford.edu>) under the BSD license
- The AVX2 popcount implementation by Daniel Lemire, Nathan Kurz, Owen Kaser, et al. under the Apache 2 license
- heapq and ascii_buffer_converter by the Python Software Foundation under the Python license
- TimSort code by Christopher Swenson under the MIT License
- tests/unittest2 by Steve Purcell, the Python Software Foundation, and others, under the Python license
- chemfp/rdmaccs.patterns and chemfp/rdmaccs2.patterns by Rational Discovery LLC, Greg Landrum, and Julie Penzotti, under the 3-Clause BSD License

CHAPTER 3

What's new in version 3.1

Released 17 September 2017

The new specialized POPCNT implementation for PubChem/CACTVS keys increases search performance for that case by about 15%.

The SearchResults object gained the `to_csr()` method and the `shape` attribute. The new method returns a SciPy compressed sparse row matrix containing the similarity scores, which can be passed into scikit-learn for clustering.

The fall 2017 release of OEChem will accept InChI strings as structure input. The chemfp wrapper now knows about this, as well as the two new InChI output flavors "RelativeStereo" and "RacemicStereo".

The fall 2017 release of RDKit will fix a bug in the pattern fingerprint definitions. The new chemfp fingerprint type is RDKit-Pattern/4.

Changed how oe2fps, rdkit2fps, and ob2fps report missing or empty identifiers. Previously the default `--errors` setting of "ignore" simply skipped those records, without any warning messages. This caused problems processing the ChEBI SD file. Most of the records have an empty title line, so only a few fingerprint records were generated. It wasn't obvious that the resulting data set was useless. The new code always reports a warning for empty or missing identifiers, even with "ignore". If the `--errors` is "strict" then the warning becomes an error and processing stops.

Updated the #software line to include "chemfp/1.3" in addition to the toolkit information. This helps distinguish between, say, two different programs which generate RDKit Morgan fingerprints. It's also possible that a chemfp bug can affect the fingerprint output, so the extra term makes it easier to identify a bad dataset.

There are several small fixes related to memory leaks, the bytes/Unicode distinction in Python 3, error messages, and error handling.

Removed chemfp.progressbar and chemfp.futures. These were included in chemfp 1.1 because I used them in a project for one customer and thought they might be useful in future chemfp projects. They were not. Also removed chemfp.argparse because chemfp 3.0 dropped support for Python 2.6.

CHAPTER 4

What's new in version 3.0.1

Released 28 August 2017

This is a bug-fix release. This fixes a critical bug in the general-purpose POPCNT popcount implementation and a bug in the code to detect the RDKit Pattern fingerprint change in 2017.3.

See the CHANGELOG for details.

CHAPTER 5

What's new in version 3.0

Released 2 May 2017

Chemfp now supports both Python 2.7 and Python 3.5 or later. It no longer supports version before Python 2.7. Chemfp will support Python 2.7 at least until 2020, which is the end-of-life for Python 2.7.

This required extensive changes to distinguish between text/Unicode strings and byte strings. The biggest user-facing change is that identifiers are now treated as Unicode strings. Fingerprints are still treated as byte strings.

This change is not backwards compatible. The APIs function parameters are polymorphic, so in most cases you can pass in either a Unicode string or a UTF-8 encoded byte string. However, the return type for an identifier is Unicode, which will likely cause problems with existing code which expects bytes.

All of the chemistry toolkits have decided to treat files as UTF-8 encoded. Chemfp's "text toolkit" offers limited support for reading Latin-1 encoded files. This is a tricky topic so contact me if you have questions or problems.

I have removed the "make_string_creator()" function because it was hard to explain, hard to maintain, and had little performance improvement over passing in the arguments to `chemfp.create_string()`. This will break compatibility, but then again, I don't think anyone used it. If it is a problem, I suggest creating a function, as in the following:

```
>>> from chemfp import rdkit_toolkit as T
>>> mol = T.parse_molecule("c1ccccc1O", "smistring")
>>> T.create_string(mol, "smistring", writer_args = {"allBondsExplicit": True})
u'O-c1:c:c:c:c:c:1'
>>> def make_string(mol):
...   return T.create_string(mol, "smistring", writer_args = {"allBondsExplicit":␣
↪True})
...
>>> make_string(mol)
u'O-c1:c:c:c:c:c:1'
```

If you look carefully at the previous example, you'll see the other major backwards incompatibility. The function `chemfp.create_string()` now return a Unicode string instead of a byte string. This also means its *format* parameter no longer accepts the ".zlib" or ".gzip" extensions.

Instead, to get the old behavior use the new API function `chemfp.create_bytes()`":

```
>>> T.create_bytes(mol, "smistring", writer_args = {"allBondsExplicit": True})
'O-c1:c:c:c:c:c:1'
>>> T.create_bytes(mol, "smistring.zlib", writer_args = {"allBondsExplicit": True})
'x\x9c\xf3\xd7M6\xb4J\x86CC\x00&\xc8\x04\x8d'
```

There's a similar change between `chemfp.open_molecule_writer_to_string()` and the new function `chemfp.open_molecule_writer_to_bytes()`.

There are also some new features in version 3.0 which don't break compatibility.

Similarity search is faster because there are now specialized popcount implementations based on the fingerprint length. On one benchmark, 166-bit searches are 35% faster, 1024-bit searches are 25% faster, and 2048-bit searches are 5% faster.

There is a new popcount implementation for processors with the AVX2 instruction set. It is about 15% faster than the POPCNT version for 2048 bit fingerprints. To test it out you will have to compile chemfp with *--with-avx2* enabled.

Added support for the Avalon fingerprints in RDKit, if RDKit has been compiled with Avalon support.

CHAPTER 6

What's new in version 2.1

Released 2 July 2015

Version 2.1 adds Tversky support for every place there was Tanimoto search (except the handful of deprecated APIs). There are new search routines for FPS and arena searches, including OpenMP support, and new bitops functions to compute a Tversky index between two fingerprints.

The k-nearest arena searches now support OpenMP. Previously they were single threaded even though the other search functions supported multiple threads.

The built-in SDF parser saw a couple improvements, including support for both "\n" and "\r\n" newlines, instead of only "\n" newlines.

There were a number of bug fixes that concern edge cases. For example, some 64-bit double calculations could be off-by-one in the last digit, and fingerprints with 0 bits set could cause a few problems.

CHAPTER 7

What's new in version 2.0

Released 8 April 2015

Version 2.0 includes many new features designed for web service development. The new "FPB" binary fingerprint file format is very fast to load, which is great for web server reloading during development and on the command-line. The speed comes from using a memory-mapped file, which also means that multiple chemfp instances can use the same file on the same machine without extra memory overhead.

The most extensive improvement is the new portable API for working with structure files and fingerprint types. The moment you start working with multiple chemistry toolkits, you realize that they all have different ways to read and write molecules, and to generate fingerprints from a molecule. Chemfp tries hard to have a consistent API for these common tasks, without sacrificing performance, so you can get get your work done. For example, with the new API it's easy to take an SD record as an input string, compute the MACCS fingerprints for each available toolkit, add the results as new SD tags, and return the updated record.

This sounds so easy, doesn't it? It took about a year to develop. The API is quite extensive, and includes the ability to pass toolkit-specific options to the underlying parsers, a low-level SDF parser that can be used to index a file, a way to get a list of available formats and fingerprint types, and methods to parse fingerprint arguments from strings.

New with version 2.0 is the ability to handle PubChem-sized data. Previous versions used 32 bit indexing and had a limit of 4GB, which is enough for 33M 1024-bit fingerprints, but PubChem has about twice that many structures.

There are also a lot of improvements, bug fixes, and performance tweaks. For example, the FPS reader is now almost twice as fast! For details, see the CHANGELOG file of the release.

CHAPTER 8

Future

The chemfp code base is solid and in use at many companies, some of whom have paid for the commercial version. It has great support for fingerprint generation, fast similarity search, and toolkit portability, but there's plenty left to do in future. Here's a mixture of things that are likely and things which are possibilties. Of course funding and feedback would help prioritize things. Let me know if you need something like one of these.

Right now you're limited to the built-in toolkit fingerprint types, plus chemfp's own SMARTS-based fingerprints. There should be a registration system so you can tell chemfp about user-defined fingerprint types.

I would like some way to select fingeprint subsets. My original thought was something like an awk for the FPS format, with the ability to select N fingerprints at random, or those matching a given set of identifiers, etc. My current thought is to implement it as a sqlite virtual table.

Chemfp supports Tanimoto and Tversky similarity. I could also add support for other measures; cosine and Hamming seem like the most useful other alternatives.

Chemfp does not currently support Microsoft Windows computer because the code assumes the LP64 model, where "int" is 32 bits and "long" is 64 bits. It will require a lot of low-level work to tweak everything correctly for the Windows LLP64 model, where "int" and "long" are 32 bits and "long long" is 64 bits. Once that's done, I'll have to figure out how to make an installer. I've decided to put it off until a someone (or someones) fund it.

The threshold and k-nearest arena search results store hits using compressed sparse rows. These work well for sparse results, but when you want the entire similarity matrix (ie, with a minimum threshold of 0.0) of a large arena, then time and space to maintain the sparse data structure becomes noticable. It's likely in that case that you want to store the scores in a 2D NumPy matrix.

I'm really interested in using chemfp to handle different sorts of clustering. Let me know if there are things I can add to the API which would help you do that.

If you are not a Python programmer then you might prefer that the core search routines be made accessible through a C API. That's possible, in that the software was designed with that in mind, but it needs more development and testing.

Chemfp ever since version 1.1 supports OpenMP. That's great for shared-memory machines. Are you interested in supporting a distributed computing version?

There are any number of higher-level tools which can be built on the chemfp components. For example, what about a wsgi component which implements a web-based search API for your local network? Wouldn't it be nice to say:

```
fpserver filename1.fpb
```

and have a simple search service?

What about an IPython visualization tool?

There's a paper (doi:10.1093/bioinformatics/byq067) on using locality-sensitive hashing to find highly similar fingerprints. Are there cases where it's more useful than chemfp's direct search?

Several people have asked about GPU implementations. My feeling is that the CPU is fast enough, and much easier to deploy. That's not saying I wouldn't be interested in a GPU implementation, only describing why it's not at the top of the list.

CHAPTER 9

Thanks

In no particular order, the following contributed to chemfp in some way: Noel O'Boyle, Geoff Hutchison, the Open Babel developers, Greg Landrum, OpenEye, Roger Sayle, Phil Evans, Evan Bolton, Wolf-Dietrich Ihlenfeldt, Rajarshi Guha, Dmitry Pavlov, Roche, Kim Walisch, Daniel Lemire, Nathan Kurz, Chris Morely, Jörg Kurt Wegner, Phil Evans, Björn Grüning, Andrew Henry, Brian McClain, Pat Walters, Brian Kelley, and Lionel Uran Landaburu.

Thanks also to my wife, Sara Marie, for her many years of support.

CHAPTER 10

Indices and tables

- genindex
- modindex
- search

Python Module Index

C

Index

Symbols

A

B

C

D

E

F

G

H

I

K

L

M

N

O

P

R

S

T

U

V

W