

---

# **chemfp Documentation**

*Release 1.3*

**Andrew Dalke**

**Sep 18, 2017**



<b>1</b>	<b>Installing</b>	<b>3</b>
1.1	Configuration options . . . . .	4
<b>2</b>	<b>Working with the command-line tools</b>	<b>5</b>
2.1	Generating fingerprint files from PubChem SD files . . . . .	5
2.2	k-nearest neighbor search . . . . .	6
2.3	Threshold search . . . . .	6
2.4	Combined k-nearest and threshold search . . . . .	7
2.5	NxN (self-similar) searches . . . . .	8
2.6	Using a toolkit to process the ChEBI dataset . . . . .	8
2.7	Alternate error handlers . . . . .	12
2.8	chemfp's two cross-toolkit substructure fingerprints . . . . .	12
<b>3</b>	<b>Help for the command-line tools</b>	<b>15</b>
3.1	ob2fps command-line options . . . . .	15
3.2	oe2fps command-line options . . . . .	16
3.3	rdkit2fps command-line options . . . . .	17
3.4	sdf2fps command-line options . . . . .	19
3.5	simsearch command-line options . . . . .	20
<b>4</b>	<b>The chemfp Python library</b>	<b>23</b>
4.1	Byte and hex fingerprints . . . . .	23
4.2	Fingerprint collections and metadata . . . . .	24
4.3	FingerprintArena . . . . .	25
4.4	How to use query fingerprints to search for similar target fingerprints . . . . .	27
4.5	How to search an FPS file . . . . .	29
4.6	FingerprintArena searches returning indices instead of ids . . . . .	30
4.7	Computing a distance matrix for clustering . . . . .	32
4.8	Convert SearchResults to a SciPy csr matrix . . . . .	33
4.9	Taylor-Butina clustering . . . . .	34
4.10	Reading structure fingerprints using a toolkit . . . . .	36
4.11	Select a random fingerprint sample . . . . .	37
4.12	Look up a fingerprint with a given id . . . . .	39
4.13	Sorting search results . . . . .	40
4.14	Working with raw scores and counts in a range . . . . .	41
<b>5</b>	<b>chemfp API</b>	<b>47</b>

<b>6</b>	<b>chemfp top-level module</b>	<b>49</b>
6.1	ChemFPError . . . . .	52
6.2	ParseError . . . . .	52
6.3	Metadata . . . . .	53
6.4	FingerprintReader . . . . .	54
6.5	FingerprintIterator . . . . .	55
6.6	Fingerprints . . . . .	55
6.7	FingerprintWriter . . . . .	56
6.8	ChemFPPProblem . . . . .	56
6.9	Open Babel fingerprints . . . . .	62
6.10	OpenEye fingerprints . . . . .	62
6.11	RDKit fingerprints . . . . .	62
<b>7</b>	<b>chemfp.arena module</b>	<b>65</b>
7.1	FingerprintArena . . . . .	65
<b>8</b>	<b>chemfp.search module</b>	<b>69</b>
8.1	SearchResults . . . . .	76
8.2	SearchResult . . . . .	78
<b>9</b>	<b>chemfp.bitops module</b>	<b>81</b>
<b>10</b>	<b>chemfp.encodings</b>	<b>83</b>
<b>11</b>	<b>chemfp.fps_io module</b>	<b>87</b>
11.1	FPSReader . . . . .	87
11.2	FPSWriter . . . . .	90
<b>12</b>	<b>chemfp.io module</b>	<b>91</b>
12.1	Location . . . . .	91
<b>13</b>	<b>License and advertisement</b>	<b>95</b>
<b>14</b>	<b>What's new in 1.3</b>	<b>97</b>
14.1	Toolkit changes . . . . .	97
14.2	Performance . . . . .	98
14.3	Command-line tools . . . . .	98
14.4	API . . . . .	98
14.5	Important bug fixes . . . . .	99
14.6	Configuration . . . . .	99
<b>15</b>	<b>Future</b>	<b>101</b>
<b>16</b>	<b>Thanks</b>	<b>103</b>
<b>17</b>	<b>Indices and tables</b>	<b>105</b>
	<b>Python Module Index</b>	<b>107</b>

`chemfp` is a set of tools for working with cheminformatics fingerprints in the FPS format.

This is the documentation for the no-cost version of chemfp. To see the documentation for the chemfp 3.1, the commercial version of chemfp, go to <http://chemfp.readthedocs.io/en/chemfp-3.1/>.

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.

The programs are built using the *chemfp Python library API*, which in turn uses a C extension for the performance critical sections. The parts of the library API documented here are meant for public use, and include examples.

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

Chemfp 1.3 was released on 18 September 2017. It supports Python 2.7 and can be used with any recent version of OEChem/OEGraphSim, Open Babel, or RDKit. Python 3 support is available in the commercial version of chemfp. If you are interested in paying for a copy, send an email to [sales@dalkescientific.com](mailto:sales@dalkescientific.com).



---

## 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?

Chemfp 1.3 only supports Python 2.7. It might work under Python 2.6 but that configuration hasn't been tested. It will not work under Python 2.5.

The core chemfp functionality (e.g. similarity search) 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.

Chemfp 1.3 has been tested with Open Babel 2.4.1, RDKit 2013.03, RDKit 2016.09, RDKit 2017.03, RDKit 2017.09 (dev), OEChem/OEGraphSim 2014.07, OEChem/OEGraphSim 2016.10, and OEChem/OEGraphSim 2017.10 (beta).

The easiest way to install chemfp is with the `pip` installer. This comes with Python 2.7.9 or later so it may already be installed. Chemfp 1.3 is available through [PyPI \(the Python Package Index\)](#) so you can install it over the web as:

```
pip install chemfp
```

To install the `tar.gz` file with `pip`:

```
pip install chemfp-1.3.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-1.3.tar.gz
cd chemfp-1.3
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-1.3.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-1.3.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 me 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 compiled that 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.

---

## 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.

### Generating fingerprint files from PubChem SD files

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 use.

Start by downloading the files `Compound_027575001_027600000.sdf.gz` (from [ftp://ftp.ncbi.nlm.nih.gov/pubchem/Compound/CURRENT-Full/SDF/Compound\\_027575001\\_027600000.sdf.gz](ftp://ftp.ncbi.nlm.nih.gov/pubchem/Compound/CURRENT-Full/SDF/Compound_027575001_027600000.sdf.gz)) and `Compound_014550001_014575000.sdf.gz` (from [ftp://ftp.ncbi.nlm.nih.gov/pubchem/Compound/CURRENT-Full/SDF/Compound\\_014550001\\_014575000.sdf.gz](ftp://ftp.ncbi.nlm.nih.gov/pubchem/Compound/CURRENT-Full/SDF/Compound_014550001_014575000.sdf.gz)). At the time of writing they contain 213 and 5208 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!

How does this work? Each PubChem record contains the precomputed CACTVS substructure keys in the `PUBCHEM_CACTVS_SUBSKEYS` tag. 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 fingerprints are the same as the order of the corresponding record in the SDF, although unconvertable records might be skipped, depending on the `--errors` flag.

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. 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 fingerprint files generated in *Generating fingerprint files from PubChem SD files* but you do not need a chemistry toolkit.

We'll use the `pubchem_queries.fps` as the queries for a `k=2` nearest neighbor similarity search of the target file `pubchem_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/1.3
#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
```

How do you 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 1455201 with score 0.7236 and the second hit is 14566941 with score 0.7105. Since this is a k-nearest neighbor 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 fingerprint files generated in *Generating fingerprint files from PubChem SD files* 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 20 lines of output from this are:

```
#Simsearch/1
#num_bits=881
#type=Tanimoto k=all threshold=0.738
#software=chemfp/1.3
#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
0 27575240
0 27575250
1 27575257      14572463      0.7468
1 27575282      14555201      0.7656
0 27575284
0 27575295
0 27575318
0 27575419
```

Take a look at the fourth result line, which contains the 3 hits for the query id 27575208. As before, the hit information alternates between the target ids and the target scores, but unlike the k-nearest search, the hits are not in a particular order. You can see that here where the scores are 0.7584, 0.7542, and 0.7701 .

You might be wondering why I chose the 0.738 threshold. Query id 27575208 has 10 hits with a threshold of 0.7 or higher. That requires 22 columns to show, which is a bit overwhelming.

## 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 [Generating fingerprint files from PubChem SD files](#) 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/1.3
#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
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
```

```

3  27575282      14555201      0.7656  14555198      0.7317  14566941      0.
↪7166
0  27575284
0  27575295
0  27575318
3  27575419      14570951      0.7339  14570934      0.7265  14570935      0.
↪7232

```

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

Use the `-NxN` option if you want to use the same fingerprints as both the queries and targets:

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

This is about twice as fast and uses half as much memory compared to:

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

Plus, 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 chemistry 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 <http://www.ebi.ac.uk/chebi/> which contains "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](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 id is stored as the value of the "ChEBI ID" tag, which in the SD file looks like:

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



```
[23:25:17] S group MUL ignored on line 103
[23:25:17] 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/1.3
#source=ChEBI_lite.sdf.gz
#date=2017-09-14T21:25:17
10208220141258c184490038b4124609db0030024a0765883c62c9e1288a1dc224de62f445743b8b
30ad542718468104d521a214227b29ba3822fbf20e15491802a051532cd10d902c39b02b51648981
9c87eb41142811026d510a890a711cb02f2090ddacd990c5240cc282090640103d0a0a8b460184f5
11114e2a8060200804529804532313bb03912d5e2857a6028960189e370100052c63474748a1c000
8079f49c484ca0c0d0bcb2c64b72401042a1f82002b097e852830e5898302021a1203e412064814
a598741c014e9210bc30ab180f0162029d4c446aa01c34850071e4ff037a60e732fd85014344f82a
344aa98398654481b003a84f201f518f CHEBI:90
00000000080200412008000008000004000010100022008000400002000020100020006000800001
01000100080001000010000002002200000200000008000000400002100000000080000004401000
8020002080020000200000140002206400000424481000000000080000a80012002020004198002
00080200020020120040203001000802010100024211000004400000000100200003000001000100
0100021000a200601080002a00002020048004030000884084000008000002040200010800000000
200001002200080000200002000140002080010002504000000200a080244000060008000000802
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.

The FPS fingerprint file format allows identifiers with a space, or comma, or anything other tab, newline, and a couple of other special 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"
```

## Generating 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 uses the 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 use:

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

## Generating 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 produce 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.

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.

## Generating 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 just under 6 minutes on my desktop, and RDKit did not generate fingerprints for 1,101 of the 95,955 records.

You can see some of the RDKit error messages in the output, like:

```
[00:47:02] Explicit valence for atom # 12 N, 4, is greater than permitted
[00:47:02] S group DAT ignored on line 102
```

These come from RDKit’s error log. RDKit is careful to check that structures make chemical sense, and in this case it didn’t like the 4-valent nitrogen. It refuses to process this molecule.

The default generates RDKit’s path fingerprints with parameters:

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

(NOTE! In chemfp 1.1 the default nBitsPerHash was 4. The RDKit default nBitsPerHash is 2.)

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 and not the separately licensed OEGraphSim add-on.)



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.

---

## Help for the command-line tools

---

### 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}] [--version]
             [filenames [filenames ...]]

Generate FPS fingerprints from a structure file using Open Babel

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         generate 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)
--version
                show program's version number and exit

```

## 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}] [--version]
              [filenames [filenames ...]]

Generate FPS 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
  --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)
  --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

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, ism, usm, can, smi.gz, ism.gz, can.gz
SDF            sdf, mol, sdf.gz, mol.gz
SKC            skc, skc.gz
CDK            cdk, cdk.gz
MOL2           mol2, mol2.gz
PDB            pdb, ent, pdb.gz, ent.gz
MacroModel    mmod, mmod.gz
OEBinary v2    oeb, oeb.gz

```

## rdkit2fps command-line options

The following comes from `rdkit2fps --help`:

```
usage: rdkit2fps [-h] [--fpSize INT] [--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|1]
                [--bitFlags INT] [--pattern] [--substruct] [--rdmaccs]
                [--rdmaccs/1] [--id-tag NAME] [--in FORMAT] [-o FILENAME]
                [--out FORMAT] [--errors {strict,report,ignore}] [--version]
                [filenames [filenames ...]]
```

Generate FPS 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 INT                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)  
 --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 bits in the fingerprint (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|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

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

File Type          Valid FORMATS (use gz if compressed)
-----
SMILES             smi, ism, usm, can, smi.gz, ism.gz, usm.gz, can.gz
SDF                sdf, mol, sd, mdl, sdf.gz, mol.gz, sd.gz, mdl.gz

```

## 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 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 eighth 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] [--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
  --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
```



---

## The chemfp Python library

---

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
- a cross-toolkit interface for reading fingerprints from a structure file

The following chapters give examples of how to use the API.

### 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 = "A"
>>> fp2 = "B"
```

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

```
>>> from chemfp import bitops
>>> bitops.byte_tanimoto(fp1, fp2)
0.33333333333333331
```

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 0.33333333333333331 when represented as a 64 bit floating point number on the computer.

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

```
>>> bitops.byte_tanimoto("apples&", "oranges")
0.5833333333333337
```

You'll get a chemfp exception if they have different lengths.

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, though you still need to know your hex digits. There are two ways to hex-encode a byte string. I suggest using chemfp's `hex_encode()` function:

```
>>> bitops.hex_encode("P1@\x84K\x1aN\x00\n\x01\xa6\x10\x98\\\x10\x11")
'503140844b1a4e000a01a610985c1011'
```

Older versions of chemfp recommended using the `s.encode()` method of strings:

```
>>> "P1@\x84K\x1aN\x00\n\x01\xa6\x10\x98\\\x10\x11".encode("hex")
'503140844b1a4e000a01a610985c1011'
```

However, this will not work on Python 3. That version of Python distinguishes between text/Unicode strings and byte strings. There is no “hex” encoding for text strings, and byte strings do not implement the “encode()” method.

Use chemfp's `hex_decode()` function to decode a hex string to a fingerprint byte string.

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.)

## Fingerprint collections and metadata

In this section you'll learn the basic operations on a fingerprint collection and the fingerprint metadata.

A fingerprint record is the fingerprint plus an identifier. In chemfp, a fingerprint collection is a object which contains fingerprint records and which follows the common API providing access to those records.

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 *Generating fingerprint files from PubChem SD files* 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')
```

but I think it's easier to view it in string format, which matches the format of the FPS header:

```
>>> 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
```

All fingerprint collections support iteration. Each step of the iteration returns the fingerprint identifier and its score. Since I know the 6th record has the id 14550045, I can write a simple loop which stops with that record:

```
>>> from chemfp.bitops import hex_encode
>>> for (id, fp) in reader:
...     print id, "starts with", hex_encode(fp)[:20]
...     if id == "14550045":
...         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
14550011 starts with 030e1c10000000000000
14550044 starts with 0f3e1c00000000000000
14550045 starts with 071e8c03000000000000
```

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

## FingerprintArena

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

The `FPSReader` reads through or searches a fingerprint file once. If you want to read the file again you have to reopen it.

Reading from disk is slow, and the FPS format is designed for ease-of-use and not performance. If you want to do many queries then it's best to store everything in memory. The `FingerprintArena` is a fingerprint collection which does that.

Here's how to load fingerprints into an arena:

```
>>> 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
```

This 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 == "14550509":
...         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
14550509 with popcount 8
```

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()` reorders the fingerprints into a data structure which is faster to search, although you can disable that 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, as in:

```
>>> len(arena)
5208
>>> arena.ids[:5]
['14550474', '14574228', '14574262', '14574264', '14574265']
>>> id, fp = arena[6]
>>> id
'14574635'
>>> arena[-1][0]
'14564974'
>>> bitops.byte_popcount(arena[-1][1])
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 (the last subarena may have fewer elements), and print information about each subarena.:

```
>>> for subarena in arena.iter_arenas(1000):
...     print subarena.ids[0], len(subarena)
...
14550474 1000
14566892 1000
14557014 1000
14562813 1000
14551392 1000
```

```
14566324 208
>>> arena[0][0]
'14550474'
>>> arena[1000][0]
'14566892'
```

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.

Thus, the FPSReaders – and all chemfp fingerprint collections – also support the `iter_arenas()` interface. Here's an example of reading the targets file 25 records at a time:

```
>>> queries = chemfp.open("pubchem_queries.fps")
>>> for arena in queries.iter_arenas(25):
...     print len(arena)
...
25
25
25
25
25
25
25
25
25
25
13
```

Those add up to 213, which you can verify is the number of structures in the original source file.

If you have a *FingerprintArena* instance 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
['27599704', '27584176', '27584181', '27593039', '27575997']
>>> queries.ids[10:15]
['27599704', '27584176', '27584181', '27593039', '27575997']
```

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.

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

```
>>> queries[10:15][1:3].ids
['27599118', '27599120']
>>> queries.ids[11:13]
['27599118', '27599120']
```

## 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.

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

```
>>> import chemfp
>>> targets = chemfp.load_fingerprints("pubchem_targets.fps")
>>> len(targets)
5208
```

and open the queries as an FPSReader.

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

I'll use `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 [(4278, 0.7105263157894737), (4310, 0.7068062827225131)]
27575192 2 [(4269, 0.7157894736842105), (4814, 0.7114427860696517)]
27575198 4 [(4286, 0.703125), (4718, 0.7258883248730964)]
27575208 10 [(3186, 0.7108433734939759), (3881, 0.7102272727272727)]
27575240 2 [(4278, 0.7015706806282722), (4814, 0.715)]
# ... many lines omitted ...
```

I'm only showing the first two hits for the sake of space. It seems rather pointless, after all, to show all 10 hits of query id 27575198.

What you don't see is that the implementation uses the `iter_arenas()` interface on the queries so that it processes only a 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 care about is the count of the hits within 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
27575240 2
27575250 2
27575257 15
27575282 5
# ... 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, list(hits)
...
27575190 [(4814, 0.7236180904522613), (4278, 0.7105263157894737)]
27575192 [(4269, 0.7157894736842105), (4814, 0.7114427860696517)]
```

```

27575198 [(4814, 0.7286432160804021), (4718, 0.7258883248730964)]
27575208 [(4814, 0.7700534759358288), (4278, 0.7584269662921348)]
27575240 [(4814, 0.715), (4278, 0.7015706806282722)]
27575250 [(4269, 0.7127659574468085), (4814, 0.7085427135678392)]
27575257 [(3186, 0.7467532467532467), (3476, 0.725)]
27575282 [(4814, 0.765625), (5048, 0.7317073170731707)]
# ... 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.

The previous example loaded the fingerprints into a FingerprintArena. That's the fastest way to do multiple searches. Sometimes though 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 will processing the file.

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

```
>>> query_arena = next(chemfp.open("pubchem_queries.fps").iter_arenas(2))
```

This line opens the file, iterates over its fingerprint records, and return the two as an arena. Perhaps a slightly less confusing way to write the above is:

```
>>> for query_arena in chemfp.open("pubchem_queries.fps").iter_arenas(1):
...     break
```

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:
...         print "", hit
...
Hits for 27575190
('14555201', 0.7236180904522613)
('14566941', 0.7105263157894737)
('14566938', 0.7068062827225131)
('14555198', 0.6933962264150944)
('14550456', 0.675531914893617)
Hits for 27575192
('14555203', 0.7157894736842105)
('14555201', 0.7114427860696517)
('14566941', 0.6979166666666666)
('14566938', 0.694300518134715)
('14560418', 0.6927083333333334)

```

Remember that the FPSReader is based on reading an FPS file. Once you've done a search, the file is read, and you can't do another search. You'll need to reopen the file.

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 a FPSReader search and loading the fingerprints into a FingerprintArena occurs with under 10 queries, so there should be little reason to worry about this.

## 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 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:

1. Count the number of hits:
  - `chemfp.search.count_tanimoto_hits_fp()` - search an arena using a single fingerprint
  - `chemfp.search.count_tanimoto_hits_arena()` - search an arena using an arena
  - `chemfp.search.count_tanimoto_hits_symmetric()` - search an arena using itself
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
  - `chemfp.search.threshold_tanimoto_search_arena()` - search an arena using an arena
  - `chemfp.search.threshold_tanimoto_search_symmetric()` - search an arena using itself
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
  - `chemfp.search.knearest_tanimoto_search_arena()` - search an arena using an arena
  - `chemfp.search.knearest_tanimoto_search_symmetric()` - search an arena using itself

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.

```
>>> 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 `threshold_tanimoto_search_arena` search finds the target fingerprints which have a similarity score of at least 0.7 compared to the query.

You can iterate over the results 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]
...
4 [('27580389', 1.0), ('27580394', 0.8823529411764706), ('27581637', 0.75)]
2 [('27584917', 1.0), ('27585106', 0.8991596638655462)]
2 [('27584917', 0.8991596638655462), ('27585106', 1.0)]
3 [('27580389', 0.8823529411764706), ('27580394', 1.0), ('27581637', 0.
↪7094594594594594)]
16 [('27599061', 1.0), ('27599092', 0.9453125), ('27599082', 0.9090909090909091)]
```

This result is like what you saw earlier, except that it doesn't have the query id. You can get that from the arena's *id* attribute, which contains the list of fingerprint identifiers.

```
>>> 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 27580389
 ('27580389', 1.0)
 ('27580394', 0.8823529411764706)
 ('27581637', 0.75)
Hits for 27584917
 ('27584917', 1.0)
 ('27585106', 0.8991596638655462)
Hits for 27585106
 ...
```

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 is a *SearchResults* instance, with methods that include *SearchResults.get\_indices\_and\_scores()*, *SearchResults.get\_ids()*, and *SearchResults.get\_scores()*:

```
>>> for hits in results:
...     print len(hits), hits.get_indices_and_scores()[:3]
...
4 [(0, 1.0), (3, 0.8823529411764706), (15, 0.75)]
2 [(1, 1.0), (2, 0.8991596638655462)]
2 [(1, 0.8991596638655462), (2, 1.0)]
3 [(0, 0.8823529411764706), (3, 1.0), (15, 0.7094594594594594)]
16 [(4, 1.0), (8, 0.9453125), (9, 0.9090909090909091)]
>>>
>>> targets.ids[0]
'27580389'
>>> targets.ids[3]
'27580394'
>>> targets.ids[15]
'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.

```
>>> for row in results.iter_indices_and_scores():
...     print len(row), row[:3]
...
4 [(0, 1.0), (3, 0.8823529411764706), (15, 0.75)]
```

```

2 [(1, 1.0), (2, 0.8991596638655462)]
2 [(1, 0.8991596638655462), (2, 1.0)]
3 [(0, 0.8823529411764706), (3, 1.0), (15, 0.7094594594594594)]
16 [(4, 1.0), (8, 0.9453125), (9, 0.9090909090909091)]

```

This was added to get a bit more performance out of chemfp and because the API is sometimes cleaner one way and sometimes cleaner than 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.

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. Right now you can use the following to build a distance matrix and pass that to one of those tools.

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 gives a factor of two memory gain as well.

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

```

Once you’ve computed the distance matrix, clustering is easy. I installed the `hcluster` package, as well as `matplotlib`, then ran the following to see the hierarchical clustering:

```

import chemfp
import hcluster # Clustering package from http://code.google.com/p/scipy-cluster/

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

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

linkage = hcluster.linkage(distances, method="single", metric="euclidean")

# Plot using matplotlib, which you must have installed
hcluster.dendrogram(linkage, labels=dataset.ids)

import pylab
pylab.show()

```

In practice you'll almost certainly want to use one of the [scikit-learn clustering algorithms](#).

## 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](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:

```
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 relevant 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 in peace, it took 7 minutes to compute the same benchmark. Where did the roughly 16-fold performance boost come from? Money. After 1.0 was released, Roche funded me to add various optimizations, including taking advantage of the symmetry (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!

## Reading structure fingerprints using a toolkit

In this section you'll learn how to use a chemistry toolkit in order 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:

```
>>> 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=0.4):
...     print query_id, "=>",
...     for (target_id, score) in hits.get_ids_and_scores():
...         print "%s %.3f" % (target_id, score),
...     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
27575282 => CHEBI:126087 0.764 CHEBI:127676 0.764
27575284 => CHEBI:105622 0.900 CHEBI:108425 0.900
# ... many lines omitted ...
```

That's it! Pretty simple, wasn't it? You didn't even need to explicitly specify which toolkit you wanted to use.

The only new thing here is `chemfp.read_molecule_fingerprints()`. 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=RDKit-MACCS166/2
#software=RDKit/2017.09.1.dev1 chemfp/1.3
#source=ChEBI_lite.sdf.gz
#date=2017-09-14T11:19:31
```

The "type" told chemfp which toolkit to use to read molecules, and how to generate fingerprints from those molecules, while "aromaticity" told it which aromaticity model to use when reading the molecule file.

You can instead course 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.bitops import hex_encode
>>> reader = chemfp.read_molecule_fingerprints("OpenBabel-MACCS", "Compound_027575001_
↳027600000.sdf.gz")
>>> for i, (id, fp) in enumerate(reader):
...     print id, hex_encode(fp)
...     if i == 3:
...         break
...
27575433 800404000840549e848189cca1f132aedfab6eff1b
27575577 800400000000449e850581c22190022f8a8baadf1b
27575602 000000000000449e840191d820a0122eda9abaff1b
27575603 000000000000449e840191d820a0122eda9abaff1b
```

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

```
>>> from chemfp.bitops import hex_encode
>>> reader = chemfp.read_molecule_fingerprints("OpenEye-MACCS166", "Compound_
↳027575001_027600000.sdf.gz")
>>> for i, (id, fp) in enumerate(reader):
...     print id, hex_encode(fp)
...     if i == 3:
...         break
...
27575433 000000080840448e8481cdccb1f1b216daaa6a7e3b
27575577 000000080000448e850185c2219082178a8a6a5e3b
27575602 000000080000448e8401d14820a01216da983b7e3b
27575603 000000080000448e8401d14820a01216da983b7e3b
```

And if you have RDKit installed then you can do:

```
>>> from chemfp.bitops import hex_encode
>>> reader = chemfp.read_molecule_fingerprints("RDKit-MACCS166", "Compound_027575001_
↳027600000.sdf.gz")
>>> for i, (id, fp) in enumerate(reader):
...     print id, hex_encode(fp)
...     if i == 3:
...         break
...
27575433 000000000840549e84818dccb1f1323cdfab6eff1f
27575577 000000000000449e850185c22190023d8a8beadf1f
27575602 000000000000449e8401915820a0123eda98bbff1f
27575603 000000000000449e8401915820a0123eda98bbff1f
```

## 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 *FingerprintArena.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)
>>> new_subset.arena is subset.arena
False
>>> subset[7][0]
'14571646'
>>> new_subset[7][0]
'14571646'
```

The *FingerprintArena.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', '14566849', '14556313']
>>> [targets.ids[3112], targets.ids[0], targets.ids[1234]]
['14556313', '14550474', '14566849']
```

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

```
>>> my_ordering = targets.copy([3112, 0, 1234], reorder=False)
>>> my_ordering.ids
['14556313', '14550474', '14566849']
```

This interesting, in a boring sort of way. 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
```

## Look up a fingerprint with a given id

In this section you'll learn how to get a fingerprint record with a given id.

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'
```

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:

```
>>> fp_index = arena.get_index_by_id("14564126")
>>> arena.get_index_by_id("14564126")
2824
>>> arena[2824]
('14564126', '\x07\x1e\x1c\x00\x00 ... many bytes deleted ...')
```

## 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 ( $k \cdot \log(k)$ ) compared to the time needed to maintain the internal heap ( $N \cdot \log(N)$ ).

By comparison, the threshold searches return the hits in arbitrary order. Sorting takes up to  $N \cdot \log(N)$  time, which is extra work for those cases where you don't want sorted data. Use the `SearchResult.reorder()` method if you want the hits sorted in-place:

```
>>> import chemfp
>>> arena = chemfp.load_fingerprints("pubchem_queries.fps")
>>> query_fp = arena.get_fingerprint_by_id("27585812")
>>> from chemfp import search
>>> result = search.threshold_tanimoto_search_fp(query_fp, arena, threshold=0.90)
>>> len(result)
6
>>> result.get_ids_and_scores()
[('27585852', 0.901840490797546), ('27586264', 0.9024390243902439),
 ('27585812', 1.0), ('27585979', 0.9753086419753086), ('27586050',
 0.9753086419753086), ('27586369', 0.9166666666666666)]

>>> result.reorder("decreasing-score")
>>> result.get_ids_and_scores()
[('27585812', 1.0), ('27585979', 0.9753086419753086), ('27586050',
 0.9753086419753086), ('27586369', 0.9166666666666666), ('27586264',
 0.9024390243902439), ('27585852', 0.901840490797546)]

>>> result.reorder("increasing-score")
>>> result.get_ids_and_scores()
[('27585852', 0.901840490797546), ('27586264', 0.9024390243902439),
 ('27586369', 0.9166666666666666), ('27585979', 0.9753086419753086),
 ('27586050', 0.9753086419753086), ('27585812', 1.0)]
```

There are currently six different sort methods, all specified by name. 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-closest-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” option 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 are interested in other ways to help improve your clustering performance, let me know.

Each `SearchResult` has a `SearchResult.reorder()` method. If you want to reorder all of the hits of a `SearchResults` then use its `SearchResults.reorder_all()` method:

```
>>> 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]
...
>>> for query_id, row in zip(arena.ids, similarity_matrix):
...     print query_id, "->", row.get_ids_and_scores()[:3]
...
27580389 -> [('27580394', 0.8823529411764706)]
27584917 -> [('27585106', 0.8991596638655462)]
27585106 -> [('27584917', 0.8991596638655462)]
27580394 -> [('27580389', 0.8823529411764706)]
27599061 -> [('27599092', 0.9453125), ('27599082', 0.9090909090909091), ('27599303',
↳0.8461538461538461)]
27593061 -> []
...

```

It takes the same set of ordering names as `SearchResult.reorder()`.

## 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 a interval.

The length of the `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)
4720

```

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()`. The result also stores the hits, and you can get the number of hits which are within a specified interval. Here are the hits counts at or above 0.5, 0.80, and 0.95:

```
>>> result.count(0.5)
1240
>>> result.count(0.8)
9
>>> result.count(0.95)
2

```

The first parameter, `min_score`, specifies the minimum threshold. The second, `max_score`, specifies the maximum. 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)
4718
>>> result.count(max_score=0.5)
3506
```

If you work do the addition you'll realize that that 1240 + 3506 equals 4746 which is 26 elements larger than the results size of 4720. This is because the default interval uses a closed range, and there are 27 hits with a score of exactly 0.5:

```
>>> result.count(0.5, 0.5)
26
```

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, "[)")
3480
>>> result.count(0.5, None, "(]")
1240
```

at get the expected results. (A min or max of *None* means that there is respectively no lower or no upper bound.)

Now for something a bit fancier. 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. Are the coenzyme A-like structures in ChEBI more similar to the penicillin-like structures than you would expect by comparing two randomly chosen subsets? By similar, I'll use Tanimoto similarity of the “chebi\_maccs.fps” file created in the *Using a toolkit to process the ChEBI dataset* command-line tool example XXX.

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 117 structures where the fingerprint is at least 0.95 similar to coenzyme A, and “penicillin-like” as the 15 structures at least 0.90 similar to penicillin. This gives 1755 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 `SearchResults.count_all()` method, instead of looping over each `SearchResult` and calling its `SearchResult.count()`:

```
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")

# coenzyme A
coA_arena = get_neighbors_as_arena(chebi, "CHEBI:15346", threshold=0.95)
print len(coA_arena), "coenzyme A-like structures"
```

```

# penicillin
penicillin_arena = get_neighbors_as_arena(chebi, "CHEBI:17334", threshold=0.9)
print len(penicillin_arena), "penicillin-like structures"

# I'll compute a profile at different thresholds
thresholds = [0.3, 0.35, 0.4, 0.45, 0.5, 0.6, 0.7, 0.8, 0.9]

# Compare the two sets. (For this case the speed difference between a threshold
# of 0.25 and 0.0 is not noticeable, 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 117 fingerprints and the other with 15, 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 117 or 15, respectively, nearest neighbors as the set members:

```

# 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 look something like:

```

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](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.

I first learned about this approach from the *Similarity Ensemble Approach* (SEA) work of Keiser, Roth, Armbruster, Ernsberger, and Irwin. The paper is available online from <http://sea.bkslab.org/>.

That paper actually wants you to use the “raw score”. This 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 an individual result or `SearchResults.cumulative_score_all()` for the entire set of results:

```

>>> sum(row.cumulative_score(min_score=0.5, max_score=0.9)
...     for row in coA_against_penicillin_result)
224.83239025119906
>>> coA_against_penicillin_result.cumulative_score_all(min_score=0.5, max_score=0.9)
224.83239025119866

```

These also take the *interval* parameter if you don’t want the default of `[]`.

You may wonder why these two values aren’t exactly the same. Addition of floating point numbers isn’t associative. You can see that I get still different results 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])
224.83239025119875
```



## CHAPTER 5

---

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 and compressed FPS format, respectively.

This version of chemfp does not support the FPB format. Trying to use the “fpb” format will raise a `NotImplementedError`.

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.

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`

`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 FingerprintArena data structure reads all of the fingerprints and identifiers from ‘reader’ and stores them into an in-memory data structure which supports fast similarity searches.

If ‘reader’ is a string or implements “read” then the contents will be parsed with the ‘chemfp.open’ function. Otherwise it must support iteration returning (id, fingerprint) pairs. ‘metadata’ contains the metadata the arena. If not specified then ‘reader.metadata’ is used.

The loader may reorder the fingerprints for better search performance. To prevent ordering, use `reorder=False`.

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 determines the best alignment based on the fingerprint size and available popcount methods.

#### 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", or "fps.gz". "fpb" will raise a NotImplementedError*) – The file format name if the reader is a string

**Returns** FingerprintArena

`chemfp.read_structure_fingerprints` (*type*, *source=None*, *format=None*, *id\_tag=None*, *reader\_args=None*, *errors="strict"*)

Deprecated function. Please call `read_molecule_fingerprints()` instead

The function named changed in chemfp 2.0 to `read_molecule_fingerprints()` because it was a better fit to the toolkit API. Chemfp-1.3 maintains backwards compatibility with chemfp-1.1, so the function remains. It forwards the call the correct function.

#### 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.

**Returns** a FingerprintReader

```
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 a FingerprintReader 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”.

‘aromaticity’ specifies the aromaticity model, and is only appropriate for OEChem. It must be a string like “openeye” or “daylight”.

Here is an example of using fingerprints generated from structure file:

```
fp_reader = read_molecule_fingerprints("OpenBabel-FP4/1", "example.sdf.gz")
print "Each fingerprint has", fps.metadata.num_bits, "bits"
for (id, fp) in fp_reader:
    print id, fp.encode("hex")
```

### 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.

**Returns** a FingerprintReader

```
chemfp.open_fingerprint_writer(destination, metadata=None, format=None, alignment=8,
                               reorder=True, tmpdir=None, max_pool_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.

If the output format is “fps” or “fps.gz” then *destination* may be a filename, a file object, or None for stdout. The “fpb” format is not available for this version of chemfp, and function will raise a `NotImplementedError` in that case.

The parameters *alignment*, *reorder*, *tmpdir*, and *max\_spool\_size* are for FPB output and are ignored. The parameters are listed for better forwards-compatibility.

The *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 iterated 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 *chemfp.FingerprintReader* containing a metadata and a list of (id, fingerprint) pairs.

This is typically used as an adapter 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 only concrete fingerprint writer class in chemfp 1.3 is:

- `chemfp.fps_io.FPSWriter` - write an FPS file

Chemfp 2.0 and later also implement `OrderedFPBWriter` and `InputOrderFPBWriter`.

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.

## ChemFPPProblem

**class** `chemfp.ChemFPPProblem`

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 *ChemFPPProblem* 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 *ChemFPPProblem* 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 problems then this returns an empty list. Otherwise it returns a list of *ChemFPPProblem* 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 *ChemFPPProblem* 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 of size ‘arena\_size’. 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: the *FPSReader* may be used as a target but it can only process one batch, and searching a *FingerprintArena* is faster if you have more than a few queries.

### Parameters

- **queries** (*any fingerprint container*) – The query fingerprints.
- **targets** (*FingerprintArena or the slower 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** An iterator containing (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"
```

### Parameters

- **fingerprints** (*a FingerprintArena with precomputed popcount\_indices*) – The arena containing the fingerprints.
- **threshold** – 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 of size ‘arena\_size’. 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: the FPSReader may be used as a target but it can only process one batch, and searching a FingerprintArena is faster if you have more than a few queries.

### Parameters

- **queries** (*any fingerprint container*) – The query fingerprints.
- **targets** (*FingerprintArena or the slower 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 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 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)
```

#### Parameters

- **fingerprints** (*a FingerprintArena with precomputed popcount\_indices*) – The arena containing the fingerprints.
- **threshold** – 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 of size ‘arena\_size’. 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: the FPSReader may be used as a target but it can only process one batch, and searching a FingerprintArena is faster if you have more than a few queries.

### Parameters

- **queries** (*any fingerprint container*) – The query fingerprints.
- **targets** (*FingerprintArena or the slower 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. ‘hits’ contains a list of (target\_id, score) pairs, sorted by score.

`chemfp.knearest_tanimoto_search_symmetric(fingerprints, k=3, threshold=0.7)`

Find the nearest *k* 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 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",
    print ", ".join("%.2f" % x for x in hits.get_scores())
```

### 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** – The minimum score threshold.

**Returns** An iterator of (fp\_id, SearchResult) pairs, one for each fingerprint

`chemfp.get_max_threads()`

Return the maximum number of threads available.

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.

`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()`.

## 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 1.3 implements both RDMACCS-OpenBabel/1 and RDMACCS-OpenBabel/2. Version 1.1 did not have a definition for key 44.

## 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 1.3 implements both RDMACCS-OpenEye/1 and RDMACCS-OpenEye/2. Version 1.1 did not have a definition for key 44.

## RDKit fingerprints

RDKit implements nine 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
- RDKit-Avalon - RDKit's interface to the Avalon toolkit fingerprints
- RDMACCS-RDKit - a chemfp implementation of the 166 MACCS keys
- ChemFP-Substruct-RDKit - an experimental chemfp implementation of the PubChem keys

Note: chemfp 1.3 implements both RDMACCS-RDKit/1 and RDMACCS-RDKit/2. Version 1.1 did not have a definition for key 44.



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  $\leq$  `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.

`__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 iterated 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 indices 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*) – indices 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*

---

## 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 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 `ctypes` 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.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 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.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 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.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  $\text{min\_score} \leq \text{score} \leq \text{max\_score}$ . The interval "(" uses the open interval where  $\text{min\_score} < \text{score} < \text{max\_score}$ . 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  $\text{min\_score} \leq \text{score} \leq \text{max\_score}$ . The interval "(" uses the open interval where  $\text{min\_score} < \text{score} < \text{max\_score}$ . 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** – 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  $-\infty$ . The default *max\_score* of None is equivalent to  $+\infty$ .

The *interval* parameter describes the interval end conditions. The default of "[]" uses a closed interval, where  $\text{min\_score} \leq \text{score} \leq \text{max\_score}$ . The interval "()" uses the open interval where  $\text{min\_score} < \text{score} < \text{max\_score}$ . The half-open/half-closed intervals "[)" and "(]" are also supported.

**Parameters**

- **min\_score** (*a float, or None for  $-\infty$* ) – the minimum score in the range.
- **max\_score** (*a float, or None for  $+\infty$* ) – 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  $-\infty$ . The default *max\_score* of None is equivalent to  $+\infty$ .

The *interval* parameter describes the interval end conditions. The default of “[” uses a closed interval, where  $\text{min\_score} \leq \text{score} \leq \text{max\_score}$ . The interval “(” uses the open interval where  $\text{min\_score} < \text{score} < \text{max\_score}$ . 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` (*super\_fp*, *sub\_fp*)

Return 1 if the on bits of *sub\_fp* are also 1 bits in *super\_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,  $fp1 \wedge fp2$

`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_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

`chemfp.bitops.byte_popcount` (*fp*)

Return the number of bits set in a byte fingerprint

`chemfp.bitops.byte_tanimoto` (*fp1*, *fp2*)

Compute the Tanimoto similarity between two byte fingerprints

`chemfp.bitops.byte_to_bitlist` (*bitlist*)

Return a sorted list of the on-bit positions in the byte fingerprint

`chemfp.bitops.byte_union` (*fp1*, *fp2*)

Return the union of the two byte strings,  $fp1 \vee fp2$

`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,  $fp1 \wedge fp2$ . 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 fingerprint, or -1 if either string is a non-hex string

`chemfp.bitops.hex_isvalid` (*s*)  
Return 1 if the string is a valid hex fingerprint, otherwise 0

`chemfp.bitops.hex_popcount` (*fp*)  
Return the number of bits set in a hex fingerprint, 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 -1.0 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,  $fp1 \vee fp2$ . 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 releases 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, '\xa8')
>>> from_binary_lsb('111101')
(5, '\x17')
>>> from_binary_lsb('0000000000000001000000000000')
(29, '\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('10101000')
(8, '\xa8')
>>> from_binary_msb('00010101')
(8, '\x15')
>>> from_binary_msb('00111')
(5, '\x07')
>>> from_binary_msb('0000000000000100000000000000')
(29, '\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, 'Hi')
>>> from_base64("SGk=")[1].encode("hex")
'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('10f2')
(None, '\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('10f2')
(None, '\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('102f')
(None, '\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("AAADceB7sQAEAAAAAAAAAAAAAAAAAWAAAAAwAAAAAAAAAAABwAAAHwIYAAAAADA" +
...             "rBniwygJqAACqAyVyVACSBAAhhwIa+CC4ZtgIYCLB0/CUpAhgmADIYcAgAAO" +
```





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

**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*

**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*

## 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.

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 r
↳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 successfully 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 13

---

### License and advertisement

---

This program was developed by Andrew Dalke <[dalke@dalkescientific.com](mailto:dalke@dalkescientific.com)>, Andrew Dalke Scientific, AB. It is distributed free of charge 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, there are two development tracks. You can download and use the no-cost version or you can pay money to get access to the commercial version.

In both cases you get the software under the MIT license. I'll stress that: even the commercial version of chemfp is open source software. Once you have a copy there are very few restrictions on what you can do with it. (The one exception is we have signed a non-disclosure agreement which lets you evaluate the commercial version to decide if you want to pay for it.)

The current commercial version is 3.1. It can handle more than 4GB of fingerprint data, it supports the FPB binary fingerprint format for fast loading, it has an expanded API designed for web server development (for example, reading and writing from strings, not just files), and it supports both Python 2.7 and Python 3.5 or later.

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.

If you have questions about or with to purchase the commercial distribution, send an email to [sales@dalkescientific.com](mailto:sales@dalkescientific.com).

```
Copyright (c) 2010-2017 Andrew Dalke Scientific, AB (Gothenburg, 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](mailto:kim.walisch@gmail.com)>, under the MIT license
- SSSE3.2 popcount implementation by Stanford University (written by Imran S. Haque <[ihaque@cs.stanford.edu](mailto:ihaque@cs.stanford.edu)>) under the BSD license
- heapq 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
- chemfp/argparse.py by Steven J. Bethard under the Apache License 2.0
- chemfp/progressbar/ by Nilton Volpato under the LGPL 2.1 and/or BSD license
- chemfp/futures/ by Brian Quinlan under the Python license

Released 18 September 2017

This release has dropped support for Python 2.5 and Python 2.6. It has been over 7 years since Python 2.7 was released, so if you're using an older Python, perhaps it's time to upgrade?

## Toolkit changes

RDKit, OEChem, Open Babel, and CDK did not implement MACCS key 44 ("OTHER") because it wasn't defined. Then Accelrys published a white paper which defined that term. All of the toolkits have updated their implementations. The corresponding chemfp fingerprint types are RDKit-MACCS166/2, OpenEye-MACCS166/3, and OpenBabel-MACCS/2. I have also updated chemfp's own RDMACCS definitions to include key 44, and changed the versions from /1 to /2.

This release supports OEChem v2 and OEChem v2 and drops support for OEChem v1, which OpenEye replaced in 2010. It also drops support for the old OEBinary format.

Several years ago, RDKit changed its hash fingerprint algorithm. The new chemfp fingerprint type is "RDKit-Fingerprint/2".

WARNING! In chemfp 1.1 the default for the RDKit-Fingerprint setting nBitsPerHash was 4. It should have been 2 to match RDKit's own default. I have changed the default to 2, but it means that your fingerprints will likely change.

Chemfp now supports the experimental RDKit substructure fingerprint. The chemfp type name is "RDKit-Pattern". There are four known versions. RDKit-Pattern/1 is many years old, RDKit-Pattern/2 was in place for several years up to 2017, RDKit-Pattern/3 was only in the 2017.3 release, and RDKit-Pattern/4 will be in the 2017.9 release. The corresponding *rdkit2fps* flag is `--pattern`.

RDKit has an adapter to use the third-party Avalon chemistry toolkit to create substructure fingerprints. Avalon support used to require special configuration but it's now part of the standard RDKit build process. Chemfp now supports the Avalon fingerprints, as the type "RDKit-Avalon/1". The corresponding *rdkit2fps* flag is `--avalon`.

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.

## Performance

The k-nearest arena search, which is used in NxM searches, is now parallelized.

The FPS reader is now much faster. As a result, `simsearch` for a single query (which uses `-scan` mode) is about 40% faster, and the time for `chemfp.load_fingerprints()` to create an arena is about 15% faster.

Similarity search performance for the MACCS keys, on a machine which supports the POPCNT instruction, is now about 20-40% faster, depending on the type of search.

## Command-line tools

In chemfp 1.1 the default error handler for `ob2fps`, `oe2fps`, and `rdkit2fps` was “strict”. If chemfp detected that a toolkit could not parse a structure, it would print an error message and stop processing. This is not what most people wanted. They wanted the processing to keep on going.

This was possible by specifying the `--errors` values “report” or “ignore”, but that was extra work, and confusing.

In chemfp 1.3, the default `--errors` value is “ignore”, which means chemfp will ignore any problems, not report a problem, and go on to the next record.

However, if the record identifier is missing (for example, if the SD title line is blank), then this will be always be reported to `stderr` even under the “ignore” option. If `--errors` is “strict” then processing will stop if a record does not contain an identifier.

Added `--version`. (Suggested by Noel O’Boyle.)

The `ob2fps --help` now includes a description of the FP2, FP3, FP4, and MACCS options.

## API

Deprecated `read_structure_fingerprints()`. Instead, call the new function `read_molecule_fingerprints()`. Chemfp 2.0 changed the name to better fit its new toolkit API. This change in chemfp 1.3 helps improve forward compatibility.

The `chemfp.search` module implements two functions to help with substructure fingerprint screening. The function `contains_fp()` takes a query fingerprint and finds all of the target fingerprints which contain it. (A fingerprint `x` “contains” `y` if all the on-bits in `y` are also on-bits in `x`.) The function `contains_arena()` does the same screening for each fingerprint in a query arena.

The new `SearchResults.shape` attribute is a 2-element tuple where the first is the size of the query arena and the second is the size of the target arena. The new `SearchResults.to_csr()` method converts the similarity scores in the `SearchResults` to a SciPy compressed sparse row matrix. This can be passed to some of the scikit-learn clustering algorithms.

Backported the FPS reader. This fixed a number of small bugs, like reporting the wrong record line number when there was a missing terminal newline. It also added some new features like a context manager.

Backported the FPS writer from Python 3.0. While it is not hard to write an FPS file yourself, the new API should make it even easier. Among other things, it understands how to write the chemfp `Metadata` as the header and it implements a context manager. Here’s an example of using it to find fingerprints with at least 225 of the 881 bits set and save them in another file:

```
import chemfp
from chemfp import bitops
with chemfp.open("pubchem_queries.fps") as reader:
```

```

with chemfp.open_fingerprint_writer(
    "subset.fps", metadata=reader.metadata) as writer:
    for id, fp in reader:
        if bitops.byte_popcount(fp) >= 225:
            writer.write_fingerprint(id, fp)

```

The new FPS reader and writer, along with the chemistry toolkit readers, support the `Location` API as a way to get information about the internal state in the readers or writers. This is another backport from chemfp 3.0.

Backported bitops functions from chemfp 3.0. The new functions are: `hex_contains()`, `hex_contains_bit()`, `hex_intersect()`, `hex_union()`, `hex_difference()`, `byte_hex_tanimoto()`, `byte_contains_bit()`, `byte_to_bitlist()`, `byte_from_bitlist()`, `hex_to_bitlist()`, `hex_from_bitlist()`, `hex_encode()`, `hex_encode_as_bytes()`, `hex_decode()`.

The last three functions related to hex encoding and decoding are important if you want to write code which is forward compatible for Python 3. Under Python 3, the simple `fp.encode("hex")` is no longer supported. Instead, use `bitops.hex_encode("fp")`.

Note that the chemfp 1.x series is unlikely to become Python 3 compatible. For Python 3 support, consider purchasing a copy of chemfp 3.1.

## Important bug fixes

Fix: As described above, the RDKit-Fingerprint `nBitsPerHash` default changed from 4 to 2 to match the RDKit default value.

Fix: Some of the Tanimoto calculations stored intermediate values as a double. As a result of incorrectly ordered operations, some Tanimoto scores were off by 1 ulp (the last bit in the double). They are now exactly correct.

Fix: if the query fingerprint had 1 bit set and the threshold was 0.0 then the sublinear bounds for the Tanimoto searches (used when there is a popcount index) failed to check targets with 0 bits set.

Fix: If a query had 0 bits then the k-nearest code for a symmetric arena returned 0 matches, even when the threshold was 0.0. It now returns the first k targets.

Fix: There was a bug in the sublinear range checks which only occurred in the symmetric searches when the `batch_size` is larger than the number of records and with a popcount just outside of the expected range.

## Configuration

The configuration of the `--with-*` or `--without-*` options (for OpenMP and SSE3) support, can now be specified via environment variables. In the following, the value "0" means disable (same as `--without-*`) and "1" means enable (same as `--with-*`):

```

CHEMFP_OPENMP - compile for OpenMP (default: "1")
CHEMFP_SSSE3  - compile SSSE3 popcount support (default: "1")
CHEMFP_AVX2   - compile AVX2 popcount support (default: "0")

```

This makes it easier to do a "pip install" directly on the tar.gz file or use chemfp under an automated testing system like tox, even when the default options are not appropriate. For example, the default C compiler on Mac OS X doesn't support OpenMP. If you want OpenMP support then install gcc and specify it with the "CC". If you don't want OpenMP support then you can do:

```
CHEMFP_OPENMP=0 pip install chemfp-1.3.tar.gz
```

## CHAPTER 15

---

### 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 multiple cheminformatics toolkits.

There are two tracks for improvements. Most of the new feature development is done in the commercial version of chemfp. I make my living in part by selling software, and few people will pay for software they can get for free.

The chemfp 1.x series is primarily in maintenance mode. I will track changes to the fingerprint types and add any new fingerprint types which might come along. I'll also backport some of the features from the commercial version. For example, I expect chemfp 1.4 will include the text toolkit API from chemfp 2.1, and identifiers will be returned as Unicode strings instead of byte strings.

I will also accept contributions to chemfp. These must be under the MIT license or similarly unrestrictive license so I can include it in both the no-cost and commercial versions of chemfp.



## CHAPTER 16

---

### 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 17

---

### Indices and tables

---

- `genindex`
- `modindex`
- `search`



**C**

chemfp, 45  
chemfp.arena, 65  
chemfp.bitops, 81  
chemfp.encodings, 83  
chemfp.fps\_io, 85  
chemfp.io, 91  
chemfp.search, 69



## Symbols

-with-openmp, -without-openmp  
     command line option, 4  
 -with-ssse3, -without-ssse3  
     command line option, 4  
 \_\_getitem\_\_() (chemfp.arena.FingerprintArena method), 66  
 \_\_getitem\_\_() (chemfp.search.SearchResults method), 76  
 \_\_init\_\_() (chemfp.FingerprintIterator method), 55  
 \_\_init\_\_() (chemfp.Fingerprints method), 55  
 \_\_init\_\_() (chemfp.io.Location method), 92  
 \_\_iter\_\_() (chemfp.FingerprintIterator method), 55  
 \_\_iter\_\_() (chemfp.FingerprintReader method), 54  
 \_\_iter\_\_() (chemfp.arena.FingerprintArena method), 66  
 \_\_iter\_\_() (chemfp.fps\_io.FPSReader method), 87  
 \_\_iter\_\_() (chemfp.search.SearchResult method), 78  
 \_\_iter\_\_() (chemfp.search.SearchResults method), 76  
 \_\_len\_\_() (chemfp.arena.FingerprintArena method), 66  
 \_\_len\_\_() (chemfp.search.SearchResult method), 78  
 \_\_len\_\_() (chemfp.search.SearchResults method), 76  
 \_\_repr\_\_() (chemfp.Metadata method), 53  
 \_\_repr\_\_() (chemfp.io.Location method), 92  
 \_\_str\_\_() (chemfp.Metadata method), 53

## A

aromaticity (chemfp.Metadata attribute), 53

## B

byte\_contains() (in module chemfp.bitops), 81  
 byte\_contains\_bit() (in module chemfp.bitops), 81  
 byte\_difference() (in module chemfp.bitops), 81  
 byte\_from\_bitlist() (in module chemfp.bitops), 81  
 byte\_hex\_tanimoto() (in module chemfp.bitops), 81  
 byte\_intersect() (in module chemfp.bitops), 81  
 byte\_intersect\_popcount() (in module chemfp.bitops), 81  
 byte\_popcount() (in module chemfp.bitops), 81  
 byte\_tanimoto() (in module chemfp.bitops), 81  
 byte\_to\_bitlist() (in module chemfp.bitops), 81  
 byte\_union() (in module chemfp.bitops), 81

## C

category (chemfp.ChemFPPProblem attribute), 56  
 check\_fingerprint\_problems() (in module chemfp), 57  
 check\_metadata\_problems() (in module chemfp), 57  
 chemfp (module), 45  
 chemfp.arena (module), 65  
 chemfp.bitops (module), 81  
 chemfp.encodings (module), 83  
 chemfp.fps\_io (module), 85  
 chemfp.io (module), 91  
 chemfp.search (module), 69  
 ChemFPError (class in chemfp), 72  
 ChemFPPProblem (class in chemfp), 76  
 clear() (chemfp.search.SearchResult method), 78  
 clear\_all() (chemfp.search.SearchResults method), 77  
 close() (chemfp.FingerprintIterator method), 55  
 close() (chemfp.FingerprintWriter method), 56  
 close() (chemfp.fps\_io.FPSReader method), 88  
 close() (chemfp.fps\_io.FPSWriter method), 90  
 closed (chemfp.fps\_io.FPSReader attribute), 87  
 command line option  
     -with-openmp, -without-openmp, 4  
     -with-ssse3, -without-ssse3, 4  
 contains\_arena() (in module chemfp.search), 76  
 contains\_fp() (in module chemfp.search), 75  
 copy() (chemfp.arena.FingerprintArena method), 67  
 copy() (chemfp.Metadata method), 53  
 count() (chemfp.search.SearchResult method), 79  
 count\_all() (chemfp.search.SearchResults method), 77  
 count\_tanimoto\_hits() (in module chemfp), 58  
 count\_tanimoto\_hits\_arena() (chemfp.fps\_io.FPSReader method), 88  
 count\_tanimoto\_hits\_arena() (in module chemfp.search), 69  
 count\_tanimoto\_hits\_fp()  
     (chemfp.arena.FingerprintArena method), 67  
 count\_tanimoto\_hits\_fp() (chemfp.fps\_io.FPSReader method), 88

count\_tanimoto\_hits\_fp() (in module chemfp.search), 69  
 count\_tanimoto\_hits\_symmetric() (in module chemfp), 59  
 count\_tanimoto\_hits\_symmetric() (in module chemfp.search), 70  
 cumulative\_score() (chemfp.search.SearchResult method), 79  
 cumulative\_score\_all() (chemfp.search.SearchResults method), 77

## D

date (chemfp.Metadata attribute), 53  
 description (chemfp.ChemFPPProblem attribute), 56

## E

error\_level (chemfp.ChemFPPProblem attribute), 56

## F

filename (chemfp.io.Location attribute), 92  
 fill\_lower\_triangle() (in module chemfp.search), 74  
 FingerprintArena (class in chemfp.arena), 65  
 FingerprintIterator (class in chemfp), 55  
 FingerprintReader (class in chemfp), 54  
 Fingerprints (class in chemfp), 55  
 FingerprintWriter (class in chemfp), 56  
 first\_line (chemfp.io.Location attribute), 92  
 FPSReader (class in chemfp.fps\_io), 87  
 FPSWriter (class in chemfp.fps\_io), 90  
 from\_base64() (in module chemfp.encodings), 84  
 from\_binary\_lsb() (in module chemfp.encodings), 83  
 from\_binary\_msb() (in module chemfp.encodings), 83  
 from\_cactvs() (in module chemfp.encodings), 84  
 from\_daylight() (in module chemfp.encodings), 85  
 from\_destination() (chemfp.io.Location method), 92  
 from\_hex() (in module chemfp.encodings), 84  
 from\_hex\_lsb() (in module chemfp.encodings), 84  
 from\_hex\_msb() (in module chemfp.encodings), 84  
 from\_on\_bit\_positions() (in module chemfp.encodings), 85  
 from\_source() (chemfp.io.Location method), 92

## G

get\_by\_id() (chemfp.arena.FingerprintArena method), 66  
 get\_fingerprint() (chemfp.arena.FingerprintArena method), 66  
 get\_fingerprint\_by\_id() (chemfp.arena.FingerprintArena method), 66  
 get\_fingerprint\_type() (chemfp.arena.FingerprintArena method), 66  
 get\_fingerprint\_type() (chemfp.FingerprintReader method), 54  
 get\_fingerprint\_type() (chemfp.fps\_io.FPSReader method), 88

get\_ids() (chemfp.search.SearchResult method), 78  
 get\_ids\_and\_scores() (chemfp.search.SearchResult method), 79  
 get\_index\_by\_id() (chemfp.arena.FingerprintArena method), 66  
 get\_indices() (chemfp.search.SearchResult method), 78  
 get\_indices\_and\_scores() (chemfp.search.SearchResult method), 79  
 get\_max\_threads() (in module chemfp), 61  
 get\_num\_threads() (in module chemfp), 61  
 get\_scores() (chemfp.search.SearchResult method), 78

## H

hex\_contains() (in module chemfp.bitops), 81  
 hex\_contains\_bit() (in module chemfp.bitops), 82  
 hex\_decode() (in module chemfp.bitops), 82  
 hex\_difference() (in module chemfp.bitops), 82  
 hex\_encode() (in module chemfp.bitops), 82  
 hex\_encode\_as\_bytes() (in module chemfp.bitops), 82  
 hex\_from\_bitlist() (in module chemfp.bitops), 82  
 hex\_intersect() (in module chemfp.bitops), 82  
 hex\_intersect\_popcount() (in module chemfp.bitops), 82  
 hex\_isvalid() (in module chemfp.bitops), 82  
 hex\_popcount() (in module chemfp.bitops), 82  
 hex\_tanimoto() (in module chemfp.bitops), 82  
 hex\_to\_bitlist() (in module chemfp.bitops), 82  
 hex\_union() (in module chemfp.bitops), 82

## I

ids (chemfp.arena.FingerprintArena attribute), 65  
 iter\_arenas() (chemfp.arena.FingerprintArena method), 67  
 iter\_arenas() (chemfp.FingerprintReader method), 54  
 iter\_arenas() (chemfp.fps\_io.FPSReader method), 87  
 iter\_ids() (chemfp.search.SearchResult method), 78  
 iter\_ids() (chemfp.search.SearchResults method), 77  
 iter\_ids\_and\_scores() (chemfp.search.SearchResults method), 77  
 iter\_indices() (chemfp.search.SearchResults method), 76  
 iter\_indices\_and\_scores() (chemfp.search.SearchResults method), 77  
 iter\_scores() (chemfp.search.SearchResults method), 77

## K

knearest\_tanimoto\_search() (in module chemfp), 60  
 knearest\_tanimoto\_search\_arena() (chemfp.fps\_io.FPSReader method), 89  
 knearest\_tanimoto\_search\_arena() (in module chemfp.search), 74  
 knearest\_tanimoto\_search\_fp() (chemfp.arena.FingerprintArena method), 68  
 knearest\_tanimoto\_search\_fp() (chemfp.fps\_io.FPSReader method), 89

knearest\_tanimoto\_search\_fp() (in module chemfp.search), 74  
 knearest\_tanimoto\_search\_symmetric() (in module chemfp), 61  
 knearest\_tanimoto\_search\_symmetric() (in module chemfp.search), 75

## L

load\_fingerprints() (in module chemfp), 50  
 location (chemfp.fps\_io.FPSReader attribute), 87  
 location (chemfp.ParseError attribute), 52  
 Location (class in chemfp.io), 91

## M

metadata (chemfp.arena.FingerprintArena attribute), 65  
 metadata (chemfp.fps\_io.FPSReader attribute), 87  
 Metadata (class in chemfp), 53  
 mol (chemfp.io.Location attribute), 92  
 msg (chemfp.ParseError attribute), 52

## N

num\_bits (chemfp.Metadata attribute), 53  
 num\_bytes (chemfp.Metadata attribute), 53

## O

offsets (chemfp.io.Location attribute), 92  
 open() (in module chemfp), 49  
 open\_fingerprint\_writer() (in module chemfp), 51  
 output\_recno (chemfp.io.Location attribute), 92

## P

ParseError (class in chemfp), 52  
 partial\_count\_tanimoto\_hits\_symmetric() (in module chemfp.search), 70  
 partial\_threshold\_tanimoto\_search\_symmetric() (in module chemfp.search), 73

## R

read\_molecule\_fingerprints() (in module chemfp), 50  
 read\_structure\_fingerprints() (in module chemfp), 50  
 recno (chemfp.io.Location attribute), 93  
 record (chemfp.io.Location attribute), 93  
 record\_format (chemfp.io.Location attribute), 93  
 reorder() (chemfp.search.SearchResult method), 79  
 reorder\_all() (chemfp.search.SearchResults method), 78

## S

save() (chemfp.arena.FingerprintArena method), 66  
 save() (chemfp.FingerprintReader method), 54  
 save() (chemfp.fps\_io.FPSReader method), 88  
 SearchResult (class in chemfp.search), 78  
 SearchResults (class in chemfp.search), 76  
 set\_num\_threads() (in module chemfp), 61

severity (chemfp.ChemFPPProblem attribute), 56  
 shape (chemfp.search.SearchResults attribute), 76  
 software (chemfp.Metadata attribute), 53  
 sources (chemfp.Metadata attribute), 53

## T

threshold\_tanimoto\_search() (in module chemfp), 59  
 threshold\_tanimoto\_search\_arena() (chemfp.fps\_io.FPSReader method), 89  
 threshold\_tanimoto\_search\_arena() (in module chemfp.search), 72  
 threshold\_tanimoto\_search\_fp() (chemfp.arena.FingerprintArena method), 68  
 threshold\_tanimoto\_search\_fp() (chemfp.fps\_io.FPSReader method), 89  
 threshold\_tanimoto\_search\_fp() (in module chemfp.search), 71  
 threshold\_tanimoto\_search\_symmetric() (in module chemfp), 60  
 threshold\_tanimoto\_search\_symmetric() (in module chemfp.search), 72  
 to\_csr() (chemfp.search.SearchResults method), 78  
 type (chemfp.Metadata attribute), 53

## W

where() (chemfp.io.Location method), 93  
 write\_fingerprint() (chemfp.FingerprintWriter method), 56  
 write\_fingerprint() (chemfp.fps\_io.FPSWriter method), 90  
 write\_fingerprints() (chemfp.FingerprintWriter method), 56  
 write\_fingerprints() (chemfp.fps\_io.FPSWriter method), 90