RDKit Documentation

Release 2013.03.1

Greg Landrum

March 09, 2013

CONTENTS

CHAPTER

ONE

AN OVERVIEW OF THE RDKIT

1.1 What is it?

- Open source toolkit for cheminformatics
  - BSD licensed
  - Core data structures and algorithms in C++
  - Python (2.x) wrapper generated using Boost.Python
  - Java and C# wrappers generated with SWIG
  - 2D and 3D molecular operations
  - Descriptor generation for machine learning
  - Molecular database cartridge for PostgreSQL
  - Cheminformatics nodes for KNIME (distributed from the KNIME community site: http://tech.knime.org/community/rdkit)
- Operational:
  - http://www.rdkit.org
  - Supports Mac/Windows/Linux
  - Quarterly releases
  - Web presence:
    * Homepage: http://www.rdkit.org

      Documentation, links
    * Sourceforge (http://sourceforge.net/projects/rdkit)

      Mailing lists, Downloads, SVN repository
    * Google code (http://code.google.com/p/rdkit/)

      Downloads, wiki
    * Github (https://github.com/rdkit)

      Bug tracker, git repository
  - **Mailing lists at https://sourceforge.net/p/rdkit/mailman/, searchable archives available for** rdkit-discuss and rdkit-devel
- History:

  - 2000-2006: Developed and used at Rational Discovery for building predictive models for ADME, Tox, biological activity
  - June 2006: Open-source (BSD license) release of software, Rational Discovery shuts down
  - to present: Open-source development continues, use within Novartis, contributions from Novartis back to open-source version

1.2 Functionality overview

- Input/Output: SMILES/SMARTS, SDF, TDT, SLN [1], Corina mol2 [1]
- "Cheminformatics":
  - Substructure searching
  - Canonical SMILES
  - Chirality support (i.e. R/S or E/Z labeling)
  - Chemical transformations (e.g. remove matching substructures)
  - Chemical reactions
  - Molecular serialization (e.g. mol <-> text)
- 2D depiction, including constrained depiction
- 2D->3D conversion/conformational analysis via distance geometry
- UFF implementation for cleaning up structures
- Fingerprinting: Daylight-like, atom pairs, topological torsions, Morgan algorithm, "MACCS keys", etc.
- Similarity/diversity picking
- 2D pharmacophores [1]
- Gasteiger-Marsili charges
- Hierarchical subgraph/fragment analysis
- Bemis and Murcko scaffold determination
- RECAP and BRICS implementations
- Multi-molecule maximum common substructure [2]
- Feature maps
- Shape-based similarity
- Molecule-molecule alignment
- Shape-based alignment (subshape alignment [3]) [1]
- Integration with PyMOL for 3D visualization
- Functional group filtering
- Salt stripping

[1] These implementations are functional but are not necessarily the best, fastest, or most complete.
[2] Contribution from Andrew Dalke
[3] Putta, S., Eksterowicz, J., Lemmen, C. & Stanton, R. "A Novel Subshape Molecular Descriptor" *Journal of Chemical Information and Computer Sciences* **43:1623–35** (2003).

- Molecular descriptor library:
  - Topological (κ3, Balaban J, etc.)
  - Compositional (Number of Rings, Number of Aromatic Heterocycles, etc.)
  - Electrotopological state (Estate)
  - clogP, MR (Wildman and Crippen approach)
  - "MOE like" VSA descriptors
  - Feature-map vectors [4]
- Machine Learning:
  - Clustering (hierarchical)
  - Information theory (Shannon entropy, information gain, etc.)
- Tight integration with the IPython notebook and qtconsole.

1.3 The Contrib Directory

The Contrib directory, part of the standard RDKit distribution, includes code that has been contributed by members of the community.

- **LEF**: Local Environment Fingerprints

  Contains python source code from the publications:

  - 1. Vulpetti, U. Hommel, G. Landrum, R. Lewis and C. Dalvit, "Design and NMR-based screening of LEF, a library of chemical fragments with different Local Environment of Fluorine" *J. Am. Chem. Soc.* **131** (2009) 12949-12959. http://dx.doi.org/10.1021/ja905207t
  - 1. Vulpetti, G. Landrum, S. Ruedisser, P. Erbel and C. Dalvit, "19F NMR Chemical Shift Prediction with Fluorine Fingerprint Descriptor" *J. of Fluorine Chemistry* **131** (2010) 570-577. http://dx.doi.org/10.1016/j.jfluchem.2009.12.024

  Contribution from Anna Vulpetti

- **M_Kossner**:

  Contains a set of pharmacophoric feature definitions as well as code for finding molecular frameworks.

  Contribution from Markus Kossner

- **PBF**: Plane of best fit

  Contains C++ source code and sample data from the publication:

  14. (a) Firth, N. Brown, and J. Blagg, "Plane of Best Fit: A Novel Method to Characterize the Three-Dimensionality of Molecules" *Journal of Chemical Information and Modeling* **52** 2516-2525 (2012). http://pubs.acs.org/doi/abs/10.1021/ci300293f

  Contribution from Nicholas Firth

- **mmpa**: Matched molecular pairs

  Python source and sample data for an implementation of the matched-molecular pair algorithm described in the publication:

[4] Landrum, G., Penzotti, J. & Putta, S. "Feature-map vectors: a new class of informative descriptors for computational drug discovery" *Journal of Computer-Aided Molecular Design* **20:751–62** (2006).

Hussain, J., & Rea, C. "Computationally efficient algorithm to identify matched molecular pairs (MMPs) in large data sets." *Journal of chemical information and modeling* **50** 339-348 (2010). http://dx.doi.org/10.1021/ci900450m

Includes a fragment indexing algorithm from the publication:

Wagener, M., & Lommerse, J. P. "The quest for bioisosteric replacements." *Journal of chemical information and modeling* **46** 677-685 (2006).

Contribution from Jameed Hussain.

1.4 License

CHAPTER

TWO

INSTALLATION

Below a number of installation recipies is presented, with varying degree of complexity.

2.1 Ubuntu

2.1.1 Ubuntu 12.04

Installation from the repositories

RDKit is available via the Ubuntu repositories, to install:

```
sudo apt-get install python-rdkit librdkit1 rdkit-data
```

Building from source

If you want to build from source use the git/svn repos to get the code, or download a tar.gz file. First you want to install the prerequisites:

```
sudo apt-get install flex bison build-essential python-numpy cmake \
                     python-dev sqlite3 libsqlite3-dev libboost-dev \
                     libboost-python-dev libboost-regex-dev
```

Fetch the source, here as tar.gz but you could use git clone:

```
wget http://downloads.sourceforge.net/project/rdkit/rdkit/QX_20XX/RDKit_20XX_XX_X.tgz
```

Untar into /opt, or a different location of (like your home dir):

```
sudo tar xzvf RDKit_20XX_XX_X.tgz -C /opt
```

Set evironmental variables in ~/.bashrc

```
export RDBASE=/opt/RDKit_20XX_XX_X
export LD_LIBRARY_PATH=$RDBASE/lib:$LD_LIBRARY_PATH
export PYTHONPATH=$RDBASE:$PYTHONPATH
```

To build, compile and (optionally) test the code:

```
cd $RDBASE
mkdir build
cd build
```

```
cmake ..
make # try `make -j 4` to use 4 processors for compilation
make install
ctest
```

The custom build has been based on a blogpost from the OPIG.

2.2 Other Linux or Mac

The instructions below are for the Q42009 release and subsequent releases.

2.2.1 Getting Ready

- Required packages:
  - cmake. You need version 2.6 (or more recent). http://www.cmake.org if your linux distribution doesn't have an appropriate package.

    Note: It seems that v2.8 is a better bet than v2.6. It might be worth compiling your own copy of v2.8 even if v2.6 is already installed.

  - flex and bison. These are frequently already installed if you have the various pieces of the development environment installed. Note that some Redhat-based systems have an extremely ancient version of flex (v2.5.4, from 1997) installed; in order to build the RDKit on these systems you need to compile and install a more recent version. The source is available at http://flex.sourceforge.net.
  - The following are required if you are planning on using the Python wrappers
    - The python headers. This probably means that you need to install the python-dev package (or whatever it's called) for your linux distribution.
    - sqlite3. You also need the shared libraries. This may require that you install a sqlite3-dev package.
    - You need to have numpy (http://www.scipy.org/NumPy) installed.

      Note: for building with XCode4 on the MacOS – there seems to be a problem with the version of numpy that comes with XCode4. Please see below in the (see *Frequently Encountered Problems*) section for a workaround.

- Optional packages
  - If you would like to install the RDKit InChI support (first available in the Q2 2011 release), follow the instructions in $RDBASE/External/INCHI-API to get a copy of the InChI source and put it in the appropriate place.

2.2.2 Installing Boost

If your linux distribution has a boost-devel package including the python and regex libraries, you can use that and save yourself the steps below.

Note: if you *do* have a version of the boost libraries pre-installed and you want to use your own version, be careful when you build the code. We've seen at least one example on a Fedora system where cmake compiled using a user-installed version of boost and then linked against the system version. This led to segmentation faults. There is a workaround for this below in the (see *Frequently Encountered Problems*) section.

- download the boost source distribution from the boost web site
- extract the source somewhere on your machine (e.g. `/usr/local/src/boost_1_45_0`)
- build the required boost libraries:
  - `cd $BOOST`
  - If you want to use the python wrappers: `./bootstrap.sh --with-libraries=python,regex`
  - If not using the python wrappers: `./bootstrap.sh --with-libraries=regex`
  - Building on 32 bit systems: `./bjam install`
  - Building on 64 bit systems: `./bjam address-model=64 cflags=-fPIC cxxflags=-fPIC install`

  If you have any problems with this step, check the boost installation instructions.

2.2.3 Building the Code

- follow the Installing Boost instructions above.
- environment variables:
  - RDBASE: the root directory of the RDKit distribution (e.g. ~/RDKit)
  - *Linux:* LD_LIBRARY_PATH: make sure it includes $RDBASE/lib and wherever the boost shared libraries were installed
  - *Mac:* DYLD_LIBRARY_PATH: make sure it includes $RDBASE/lib and wherever the boost shared libraries were installed
  - The following are required if you are planning on using the Python wrappers: * PYTHONPATH: make sure it includes $RDBASE
- Building:
  - cd to $RDBASE
  - `mkdir build`
  - `cd build`
  - `cmake ..` : See the section below on configuring the build if you need to specify a non-default version of python or if you have boost in a non-standard location
  - `make` : this builds all libraries, regression tests, and wrappers (by default).
  - `make install`

See below for a list of [#Frequently_Encountered_Problems frequently encountered problems] and solutions.

2.2.4 Testing the Build (optional, but recommended)

- cd to $RDBASE/build and do `ctest`
- you're done!

2.2.5 Advanced

Specifying an alternate Boost installation

You need to tell cmake where to find the boost libraries and header files:

If you have put boost in /opt/local, the cmake invocation would look like:

```
cmake -DBOOST_ROOT=/opt/local ..
```

Specifying an alternate Python installation

You need to tell cmake where to find the python library it should link against and the python header files.

Here's a sample command line:

```
cmake -D PYTHON_LIBRARY=/usr/lib/python2.5/config/libpython2.5.a -D PYTHON_INCLUDE_DIR=/usr/include/p
```

The `PYTHON_EXECUTABLE` part is optional if the correct python is the first version in your PATH.

Disabling the Python wrappers

You can completely disable building of the python wrappers by setting the configuration variable RDK_BUILD_PYTHON_WRAPPERS to nil:

```
cmake -D RDK_BUILD_PYTHON_WRAPPERS= ..
```

Building the Java wrappers

Additional Requirements

- SWIG v2.0.x: http://www.swig.org
- Junit: get a copy of the junit .jar file from https://github.com/KentBeck/junit/downloads and put it in the directory `$RDBASE/External/java_lib` (you will need to create the directory) and rename it to junit.jar.

Building

- When you invoke cmake add `-D RDK_BUILD_SWIG_WRAPPERS=ON` to the arguments. For example:

  ```
  cmake -D RDK_BUILD_SWIG_WRAPPERS=ON ..
  ```

- Build and install normally using *make*. The directory `$RDBASE/Code/JavaWrappers/gmwrapper` will contain the three required files: libGraphMolWrap.so (libGraphMolWrap.jnilib on the Mac), org.RDKit.jar, and org.RDKitDoc.jar.

Using the wrappers

To use the wrappers, the three files need to be in the same directory, and that should be on your CLASSPATH and in the java.library.path. An example using jython:

```
% CLASSPATH=$CLASSPATH:$RDBASE/Code/JavaWrappers/gmwrapper/org.RDKit.jar; jython -Djava.library.path=
Jython 2.2.1 on java1.6.0_20
Type "copyright", "credits" or "license" for more information.
>>> from org.RDKit import *
>>> from java import lang
>>> lang.System.loadLibrary('GraphMolWrap')
>>> m = RWMol.MolFromSmiles('c1ccccc1')
```

```
>>> m.getNumAtoms()
6L
```

2.2.6 Frequently Encountered Problems

In each case I've replaced specific pieces of the path with `....`

Problem:

```
Linking CXX shared library libSLNParse.so
/usr/bin/ld: .../libboost_regex.a(cpp_regex_traits.o): relocation R_X86_64_32S against `std::basic_st
.../libboost_regex.a: could not read symbols: Bad value
collect2: ld returned 1 exit status
make[2]: *** [Code/GraphMol/SLNParse/libSLNParse.so] Error 1
make[1]: *** [Code/GraphMol/SLNParse/CMakeFiles/SLNParse.dir/all] Error 2
make: *** [all] Error 2
```

Solution:

Add this to the arguments when you call cmake: `-DBoost_USE_STATIC_LIBS=OFF`

more information here

Problem:

```
 .../Code/GraphMol/Wrap/EditableMol.cpp:114:   instantiated from here
 .../boost/type_traits/detail/cv_traits_impl.hpp:37: internal compiler error: in make_rtl_for_nonloca
Please submit a full bug report,
with preprocessed source if appropriate.
See <URL:http://bugzilla.redhat.com/bugzilla> for instructions.
Preprocessed source stored into /tmp/ccgSaXge.out file, please attach this to your bugreport.
make[2]: *** [Code/GraphMol/Wrap/CMakeFiles/rdchem.dir/EditableMol.cpp.o] Error 1
make[1]: *** [Code/GraphMol/Wrap/CMakeFiles/rdchem.dir/all] Error 2
make: *** [all] Error 2
```

Solution:

Add `#define BOOST_PYTHON_NO_PY_SIGNATURES` at the top of `Code/GraphMol/Wrap/EditableMol.cpp`

more information here

Problem:

Your system has a version of boost installed in /usr/lib, but you would like to force the RDKit to use a more recent one.

Solution:

This can be solved by using cmake version 2.8.3 (or more recent) and providing the `-D Boost_NO_SYSTEM_PATHS=ON` argument:

```
cmake -D BOOST_ROOT=/usr/local -D Boost_NO_SYSTEM_PATHS=ON ..
```

Problem:

Building on the Mac with XCode 4

The problem seems to be caused by the version of numpy that is distributed with XCode 4, so you need to build a fresh copy.

Solution: Get a copy of numpy and build it like this as root: as root:

```
export MACOSX_DEPLOYMENT_TARGET=10.6
export LDFLAGS="-Wall -undefined dynamic_lookup -bundle -arch x86_64"
export CFLAGS="-arch x86_64"
ln -s /usr/bin/gcc /usr/bin/gcc-4.2
ln -s /usr/bin/g++ /usr/bin/g++-4.2
python setup.py build
python setup.py install
```

Be sure that the new numpy is used in the build:

```
PYTHON_NUMPY_INCLUDE_PATH        /Library/Python/2.6/site-packages/numpy/core/include
```

and is at the beginning of the PYTHONPATH:

```
export PYTHONPATH="/Library/Python/2.6/site-packages:$PYTHONPATH"
```

Now it's safe to build boost and the RDKit.

CHAPTER

THREE

GETTING STARTED WITH THE RDKIT IN PYTHON

3.1 What is this?

This document is intended to provide an overview of how one can use the RDKit functionality from Python. It's not comprehensive and it's not a manual.

If you find mistakes, or have suggestions for improvements, please either fix them yourselves in the source document (the .rst file) or send them to the mailing list: rdkit-devel@lists.sourceforge.net

3.2 Reading and Writing Molecules

3.2.1 Reading single molecules

The majority of the basic molecular functionality is found in module `rdkit.Chem`:

```
>>> from rdkit import Chem
```

Individual molecules can be constructed using a variety of approaches:

```
>>> m = Chem.MolFromSmiles('Cc1ccccc1')
>>> m = Chem.MolFromMolFile('data/input.mol')
>>> stringWithMolData=file('data/input.mol','r').read()
>>> m = Chem.MolFromMolBlock(stringWithMolData)
```

All of these functions return a `rdkit.Chem.rdchem.Mol` object on success:

```
>>> m
<rdkit.Chem.rdchem.Mol object at 0x...>
```

or None on failure:

```
>>> m = Chem.MolFromMolFile('data/invalid.mol')
>>> m is None
True
```

An attempt is made to provide sensible error messages:

```
>>> m1 = Chem.MolFromSmiles('CO(C)C')
```

displays a message like: `[12:18:01] Explicit valence for atom # 1 O greater than permitted` and

```
>>> m2 = Chem.MolFromSmiles('c1cc1')
```

displays something like: `[12:20:41] Can't kekulize mol`. In each case the value `None` is returned:

```
>>> m1 is None
True
>>> m2 is None
True
```

3.2.2 Reading sets of molecules

Groups of molecules are read using a Supplier (for example, an `rdkit.Chem.rdmolfiles.SDMolSupplier` or a `rdkit.Chem.rdmolfiles.SmilesMolSupplier`):

```
>>> suppl = Chem.SDMolSupplier('data/5ht3ligs.sdf')
>>> for mol in suppl:
...   print mol.GetNumAtoms()
...
20
24
24
26
```

You can easily produce lists of molecules from a Supplier:

```
>>> mols = [x for x in suppl]
>>> len(mols)
4
```

or just treat the Supplier itself as a random-access object:

```
>>> suppl[0].GetNumAtoms()
20
```

A good practice is to test each molecule to see if it was correctly read before working with it:

```
>>> suppl = Chem.SDMolSupplier('data/5ht3ligs.sdf')
>>> for mol in suppl:
...   if mol is None: continue
...   print mol.GetNumAtoms()
...
20
24
24
26
```

An alternate type of Supplier, the `rdkit.Chem.rdmolfiles.ForwardSDMolSupplier` can be used to read from file-like objects:

```
>>> inf = file('data/5ht3ligs.sdf')
>>> fsuppl = Chem.ForwardSDMolSupplier(inf)
>>> for mol in fsuppl:
...   if mol is None: continue
...   print mol.GetNumAtoms()
...
20
```

```
24
24
26
```

This means that they can be used to read from compressed files:

```
>>> import gzip
>>> inf = gzip.open('data/actives_5ht3.sdf.gz')
>>> gzsuppl = Chem.ForwardSDMolSupplier(inf)
>>> ms = [x for x in gzsuppl if x is not None]
>>> len(ms)
180
```

Note that ForwardSDMolSuppliers cannot be used as random-access objects:

```
>>> fsuppl[0]
Traceback (most recent call last):
  ...
TypeError: 'ForwardSDMolSupplier' object does not support indexing
```

3.2.3 Writing molecules

Single molecules can be converted to text using several functions present in the `rdkit.Chem` module.

For example, for SMILES:

```
>>> m = Chem.MolFromMolFile('data/chiral.mol')
>>> Chem.MolToSmiles(m)
'CC(O)c1ccccc1'
>>> Chem.MolToSmiles(m,isomericSmiles=True)
'C[C@H](O)c1ccccc1'
```

Note that the SMILES provided is canonical, so the output should be the same no matter how a particular molecule is input:

```
>>> Chem.MolToSmiles(Chem.MolFromSmiles('C1=CC=CN=C1'))
'c1ccncc1'
>>> Chem.MolToSmiles(Chem.MolFromSmiles('c1cccnc1'))
'c1ccncc1'
>>> Chem.MolToSmiles(Chem.MolFromSmiles('n1ccccc1'))
'c1ccncc1'
```

If you'd like to have the Kekule form of the SMILES, first Kekulize the molecule, then use the "kekuleSmiles" option:

```
>>> Chem.Kekulize(m)
>>> Chem.MolToSmiles(m,kekuleSmiles=True)
'CC(O)C1=CC=CC=C1'
```

Note: as of this writing (Aug 2008), the smiles provided when one requests kekuleSmiles are not canonical. The limitation is not in the SMILES generation, but in the kekulization itself.

MDL Mol blocks are also available:

```
>>> m2 = Chem.MolFromSmiles('C1CCC1')
>>> print Chem.MolToMolBlock(m2)

     RDKit

  4  4  0  0  0  0  0  0  0  0999 V2000
```

```
    0.0000    0.0000    0.0000 C   0  0  0  0  0  0  0  0  0  0  0  0
    0.0000    0.0000    0.0000 C   0  0  0  0  0  0  0  0  0  0  0  0
    0.0000    0.0000    0.0000 C   0  0  0  0  0  0  0  0  0  0  0  0
    0.0000    0.0000    0.0000 C   0  0  0  0  0  0  0  0  0  0  0  0
  1  2  1  0
  2  3  1  0
  3  4  1  0
  4  1  1  0
M  END
```

To include names in the mol blocks, set the molecule's "_Name" property:

```
>>> m2.SetProp("_Name","cyclobutane")
>>> print Chem.MolToMolBlock(m2)
cyclobutane
     RDKit

  4  4  0  0  0  0  0  0  0  0999 V2000
    0.0000    0.0000    0.0000 C   0  0  0  0  0  0  0  0  0  0  0  0
    0.0000    0.0000    0.0000 C   0  0  0  0  0  0  0  0  0  0  0  0
    0.0000    0.0000    0.0000 C   0  0  0  0  0  0  0  0  0  0  0  0
    0.0000    0.0000    0.0000 C   0  0  0  0  0  0  0  0  0  0  0  0
  1  2  1  0
  2  3  1  0
  3  4  1  0
  4  1  1  0
M  END
```

It's usually preferable to have a depiction in the Mol block, this can be generated using functionality in the `rdkit.Chem.AllChem` module (see the Chem vs AllChem section for more information).

You can either include 2D coordinates (i.e. a depiction):

```
>>> from rdkit.Chem import AllChem
>>> AllChem.Compute2DCoords(m2)
0
>>> print Chem.MolToMolBlock(m2)
cyclobutane
     RDKit          2D

  4  4  0  0  0  0  0  0  0  0999 V2000
    1.0607   -0.0000    0.0000 C   0  0  0  0  0  0  0  0  0  0  0  0
   -0.0000   -1.0607    0.0000 C   0  0  0  0  0  0  0  0  0  0  0  0
   -1.0607    0.0000    0.0000 C   0  0  0  0  0  0  0  0  0  0  0  0
    0.0000    1.0607    0.0000 C   0  0  0  0  0  0  0  0  0  0  0  0
  1  2  1  0
  2  3  1  0
  3  4  1  0
  4  1  1  0
M  END
```

Or you can add 3D coordinates by embedding the molecule:

```
>>> AllChem.EmbedMolecule(m2)
0
>>> AllChem.UFFOptimizeMolecule(m2)
0
>>> print Chem.MolToMolBlock(m2)
cyclobutane
     RDKit          3D
```

```
  4  4  0  0  0  0  0  0  0  0999 V2000
   -0.7931    0.5732   -0.2708 C   0  0  0  0  0  0  0  0  0  0  0  0
   -0.3802   -0.9196   -0.2340 C   0  0  0  0  0  0  0  0  0  0  0  0
    0.7838   -0.5392    0.6548 C   0  0  0  0  0  0  0  0  0  0  0  0
    0.3894    0.8856    0.6202 C   0  0  0  0  0  0  0  0  0  0  0  0
  1  2  1  0
  2  3  1  0
  3  4  1  0
  4  1  1  0
M  END
```

The optimization step isn't necessary, but it substantially improves the quality of the conformation.

If you'd like to write the molecules to a file, use Python file objects:

```
>>> print >>file('data/foo.mol','w+'),Chem.MolToMolBlock(m2)
>>>
```

3.2.4 Writing sets of molecules

Multiple molecules can be written to a file using an `rdkit.Chem.rdmolfiles.SDWriter` object:

```
>>> w = Chem.SDWriter('data/foo.sdf')
>>> for m in mols: w.write(m)
...
>>>
```

An SDWriter can also be initialized using a file-like object:

```
>>> from StringIO import StringIO
>>> sio = StringIO()
>>> w = Chem.SDWriter(sio)
>>> for m in mols: w.write(m)
...
>>> w.flush()
>>> print sio.getvalue()
mol-295
     RDKit          3D

 20 22  0  0  0  0  0  0  0  0999 V2000
    2.3200    0.0800   -0.1000 C   0  0  0  0  0  0  0  0  0  0  0  0
    1.8400   -1.2200    0.1200 C   0  0  0  0  0  0  0  0  0  0  0  0
...
  1  3  1  0
  1  4  1  0
  2  5  1  0
M  END
$$$$
```

Other available Writers include the `rdkit.Chem.rdmolfiles.SmilesWriter` and the `rdkit.Chem.rdmolfiles.TDTWriter`.

3.3 Working with Molecules

3.3.1 Looping over Atoms and Bonds

Once you have a molecule, it's easy to loop over its atoms and bonds:

```
>>> m = Chem.MolFromSmiles('C1OC1')
>>> for atom in m.GetAtoms():
...   print atom.GetAtomicNum()
...
6
8
6
>>> print m.GetBonds()[0].GetBondType()
SINGLE
```

You can also request individual bonds or atoms:

```
>>> m.GetAtomWithIdx(0).GetSymbol()
'C'
>>> m.GetAtomWithIdx(0).GetExplicitValence()
2
>>> m.GetBondWithIdx(0).GetBeginAtomIdx()
0
>>> m.GetBondWithIdx(0).GetEndAtomIdx()
1
>>> m.GetBondBetweenAtoms(0,1).GetBondType()
rdkit.Chem.rdchem.BondType.SINGLE
```

Atoms keep track of their neighbors:

```
>>> atom = m.GetAtomWithIdx(0)
>>> [x.GetAtomicNum() for x in atom.GetNeighbors()]
[8, 6]
>>> len(x.GetBonds())
2
```

3.3.2 Ring Information

Atoms and bonds both carry information about the molecule's rings:

```
>>> m = Chem.MolFromSmiles('OC1C2C1CC2')
>>> m.GetAtomWithIdx(0).IsInRing()
False
>>> m.GetAtomWithIdx(1).IsInRing()
True
>>> m.GetAtomWithIdx(2).IsInRingSize(3)
True
>>> m.GetAtomWithIdx(2).IsInRingSize(4)
True
>>> m.GetAtomWithIdx(2).IsInRingSize(5)
False
>>> m.GetBondWithIdx(1).IsInRingSize(3)
True
>>> m.GetBondWithIdx(1).IsInRing()
True
```

But note that the information is only about the smallest rings:

```
>>> m.GetAtomWithIdx(1).IsInRingSize(5)
False
```

More detail about the smallest set of smallest rings (SSSR) is available:

```
>>> ssr = Chem.GetSymmSSSR(m)
>>> len(ssr)
2
>>> list(ssr[0])
[1, 2, 3]
>>> list(ssr[1])
[4, 5, 2, 3]
```

As the name indicates, this is a symmetrized SSSR; if you are interested in the number of "true" SSSR, use the GetSSSR function.

```
>>> Chem.GetSSSR(m)
2
```

The distinction between symmetrized and non-symmetrized SSSR is discussed in more detail below in the section The SSSR Problem.

For more efficient queries about a molecule's ring systems (avoiding repeated calls to Mol.GetAtomWithIdx), use the `rdkit.Chem.rdchem.RingInfo` class:

```
>>> m = Chem.MolFromSmiles('OC1C2C1CC2')
>>> ri = m.GetRingInfo()
>>> ri.NumAtomRings(0)
0
>>> ri.NumAtomRings(1)
1
>>> ri.NumAtomRings(2)
2
>>> ri.IsAtomInRingOfSize(1,3)
True
>>> ri.IsBondInRingOfSize(1,3)
True
```

3.3.3 Modifying molecules

Normally molecules are stored in the RDKit with the hydrogen atoms implicit (e.g. not explicitly present in the molecular graph. When it is useful to have the hydrogens explicitly present, for example when generating or optimizing the 3D geometry, the `rdkit.Chem.rdmolops.AddHs` function can be used:

```
>>> m=Chem.MolFromSmiles('CCO')
>>> m.GetNumAtoms()
3
>>> m2 = Chem.AddHs(m)
>>> m2.GetNumAtoms()
9
```

The Hs can be removed again using the `rdkit.Chem.rdmolops.RemoveHs` function:

```
>>> m3 = Chem.RemoveHs(m2)
>>> m3.GetNumAtoms()
3
```

RDKit molecules are usually stored with the bonds in aromatic rings having aromatic bond types. This can be changed with the `rdkit.Chem.rdmolops.Kekulize` function:

```
>>> m = Chem.MolFromSmiles('c1ccccc1')
>>> m.GetBondWithIdx(0).GetBondType()
rdkit.Chem.rdchem.BondType.AROMATIC
>>> Chem.Kekulize(m)
>>> m.GetBondWithIdx(0).GetBondType()
rdkit.Chem.rdchem.BondType.DOUBLE
>>> m.GetBondWithIdx(1).GetBondType()
rdkit.Chem.rdchem.BondType.SINGLE
```

The bonds are still marked as being aromatic:

```
>>> m.GetBondWithIdx(1).GetIsAromatic()
True
```

and can be restored to the aromatic bond type using the `rdkit.Chem.rdmolops.SanitizeMol` function:

```
>>> Chem.SanitizeMol(m)
rdkit.Chem.rdmolops.SanitizeFlags.SANITIZE_NONE
>>> m.GetBondWithIdx(0).GetBondType()
rdkit.Chem.rdchem.BondType.AROMATIC
```

The value returned by *SanitizeMol()* indicates that no problems were encountered.

3.3.4 Working with 2D molecules: Generating Depictions

The RDKit has a library for generating depictions (sets of 2D) coordinates for molecules. This library, which is part of the AllChem module, is accessed using the `rdkit.Chem.rdDepictor.Compute2DCoords` function:

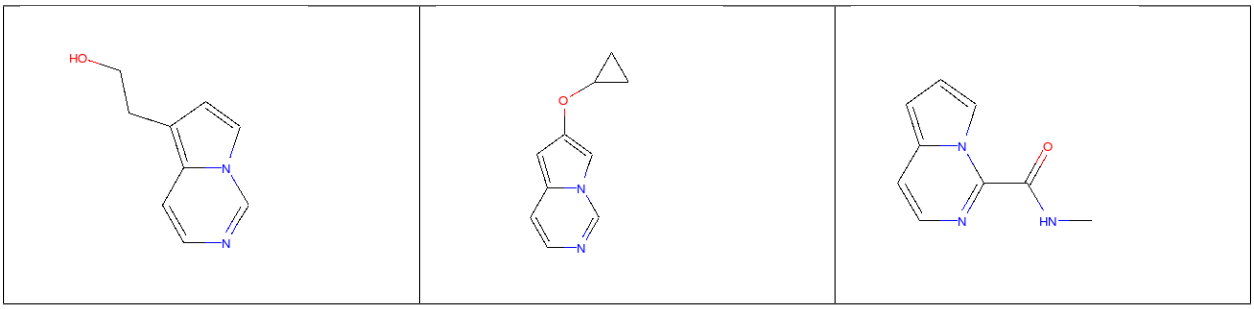
```
>>> m = Chem.MolFromSmiles('c1nccc2n1ccc2')
>>> AllChem.Compute2DCoords(m)
0
```

The 2D conformation is constructed in a canonical orientation and is built to minimize intramolecular clashes, i.e. to maximize the clarity of the drawing.

If you have a set of molecules that share a common template and you'd like to align them to that template, you can do so as follows:

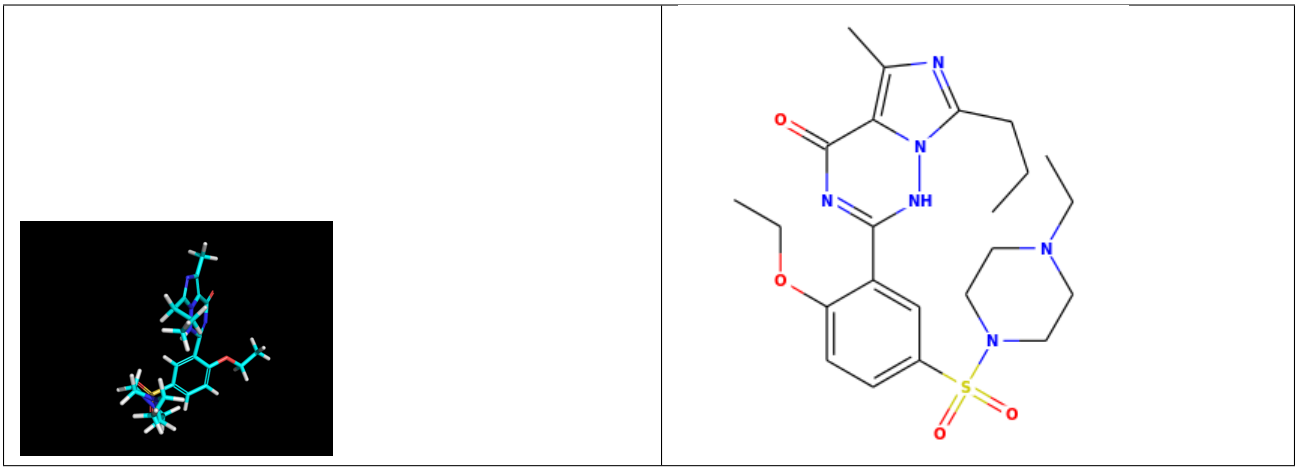
```
>>> template = Chem.MolFromSmiles('c1nccc2n1ccc2')
>>> AllChem.Compute2DCoords(template)
0
>>> AllChem.GenerateDepictionMatching2DStructure(m,template)
```

Running this process for a couple of other molecules gives the following depictions:



Another option for Compute2DCoords allows you to generate 2D depictions for molecules that closely mimic 3D conformations. This is available using the function `rdkit.Chem.AllChem.GenerateDepictionMatching3DStructure`.

Here is an illustration of the results using the ligand from PDB structure 1XP0:



More fine-grained control can be obtained using the core function `rdkit.Chem.rdDepictor.Compute2DCoordsMimicDistm` but that is beyond the scope of this document. See the implementation of GenerateDepictionMatching3DStructure in AllChem.py for an example of how it is used.

3.3.5 Working with 3D Molecules

The RDKit can generate conformations for molecules using distance geometry. [1] The algorithm followed is:

1. The molecule's distance bounds matrix is calculated based on the connection table and a set of rules.
2. The bounds matrix is smoothed using a triangle-bounds smoothing algorithm.
3. A random distance matrix that satisfies the bounds matrix is generated.
4. This distance matrix is embedded in 3D dimensions (producing coordinates for each atom).
5. The resulting coordinates are cleaned up somewhat using a crude force field and the bounds matrix.

Multiple conformations can be generated by repeating steps 4 and 5 several times, using a different random distance matrix each time.

Note that the conformations that result from this procedure tend to be fairly ugly. They should be cleaned up using a force field. This can be done within the RDKit using its implementation of the Universal Force Field (UFF). [2]

The full process of embedding and optimizing a molecule is easier than all the above verbiage makes it sound:

```
>>> m = Chem.MolFromSmiles('C1CCC1OC')
>>> m2=Chem.AddHs(m)
>>> AllChem.EmbedMolecule(m2)
0
>>> AllChem.UFFOptimizeMolecule(m2)
0
```

[1] Blaney, J. M.; Dixon, J. S. "Distance Geometry in Molecular Modeling". *Reviews in Computational Chemistry*; VCH: New York, 1994.

[2] Rappé, A. K.; Casewit, C. J.; Colwell, K. S.; Goddard III, W. A.; Skiff, W. M. "UFF, a full periodic table force field for molecular mechanics and molecular dynamics simulations". *J. Am. Chem. Soc.* **114**:10024-35 (1992) .

Disclaimer/Warning: Conformation generation is a difficult and subtle task. The 2D->3D conversion provided within the RDKit is not intended to be a replacement for a "real" conformational analysis tool; it merely provides quick 3D structures for cases when they are required.

3.3.6 Preserving Molecules

Molecules can be converted to and from text using Python's pickling machinery:

```
>>> m = Chem.MolFromSmiles('c1ccncc1')
>>> import cPickle
>>> pkl = cPickle.dumps(m)
>>> type(pkl)
<type 'str'>
>>> m2=cPickle.loads(pkl)
>>> Chem.MolToSmiles(m2)
'c1ccncc1'
```

The RDKit pickle format is fairly compact and it is much, much faster to build a molecule from a pickle than from a Mol file or SMILES string, so storing molecules you will be working with repeatedly as pickles can be a good idea.

The raw binary data that is encapsulated in a pickle can also be directly obtained from a molecule:

```
>>> binStr = m.ToBinary()
```

This can be used to reconstruct molecules using the Chem.Mol constructor:

```
>>> m2 = Chem.Mol(binStr)
>>> Chem.MolToSmiles(m2)
'c1ccncc1'
>>> len(binStr)
123
>>> len(pkl)
475
```

Note that this huge difference in text length is because we didn't tell python to use its most efficient representation of the pickle:

```
>>> pkl = cPickle.dumps(m,2)
>>> len(pkl)
157
```

The small overhead associated with python's pickling machinery normally doesn't end up making much of a difference for collections of larger molecules (the extra data associated with the pickle is independent of the size of the molecule, while the binary string increases in length as the molecule gets larger).

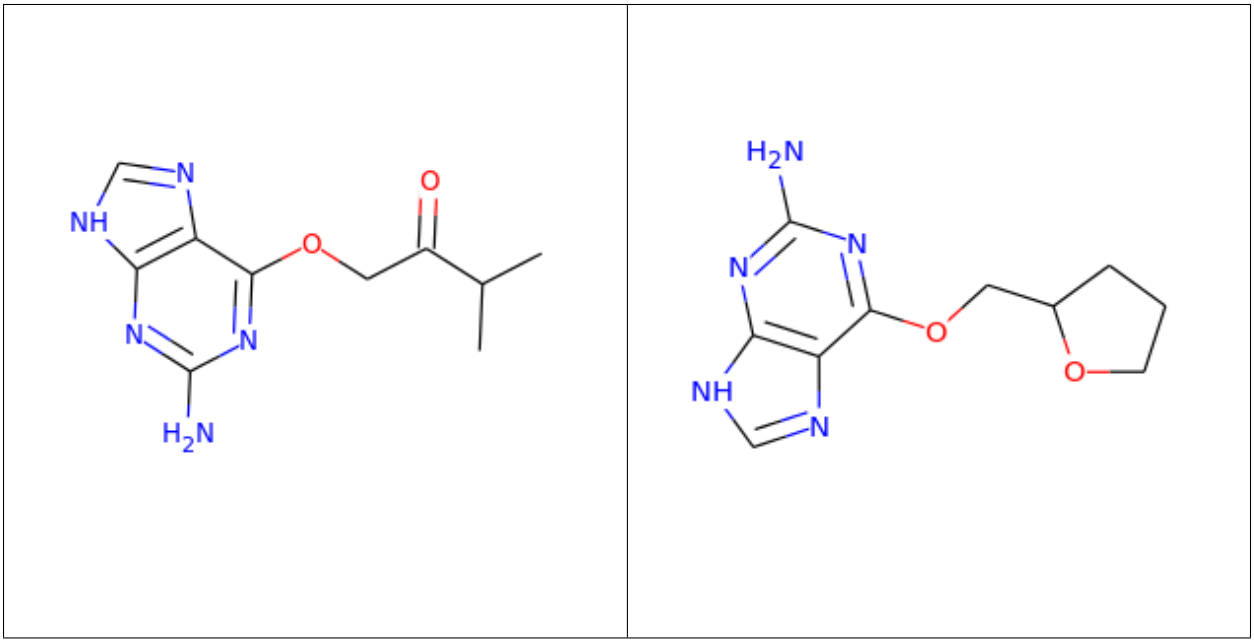
Tip: The performance difference associated with storing molecules in a pickled form on disk instead of constantly reparsing an SD file or SMILES table is difficult to overstate. In a test I just ran on my laptop, loading a set of 699 drug-like molecules from an SD file took 10.8 seconds; loading the same molecules from a pickle file took 0.7 seconds. The pickle file is also smaller – 1/3 the size of the SD file – but this difference is not always so dramatic (it's a particularly fat SD file).

3.3.7 Drawing Molecules

The RDKit has some built-in functionality for creating images from molecules found in the `rdkit.Chem.Draw` package:

```
>>> suppl = Chem.SDMolSupplier('data/cdk2.sdf')
>>> ms = [x for x in suppl if x is not None]
>>> for m in ms: tmp=AllChem.Compute2DCoords(m)
>>> from rdkit.Chem import Draw
>>> Draw.MolToFile(ms[0],'images/cdk2_mol1.png')
>>> Draw.MolToFile(ms[1],'images/cdk2_mol2.png')
```

Producing these images:



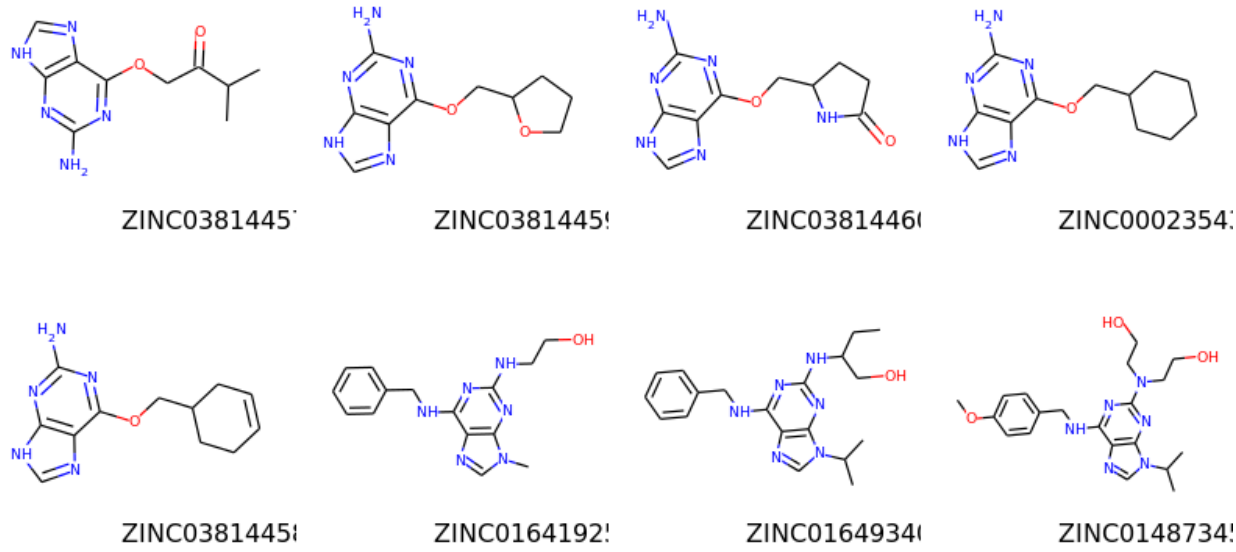
It's also possible to produce an image grid out of a set of molecules:

```
>>> img=Draw.MolsToGridImage(ms[:8],molsPerRow=4,subImgSize=(200,200),legends=[x.GetProp("_Name") for
```

This returns a PIL image, which can then be saved to a file:

```
>>> img.save('images/cdk2_molgrid.png')
```

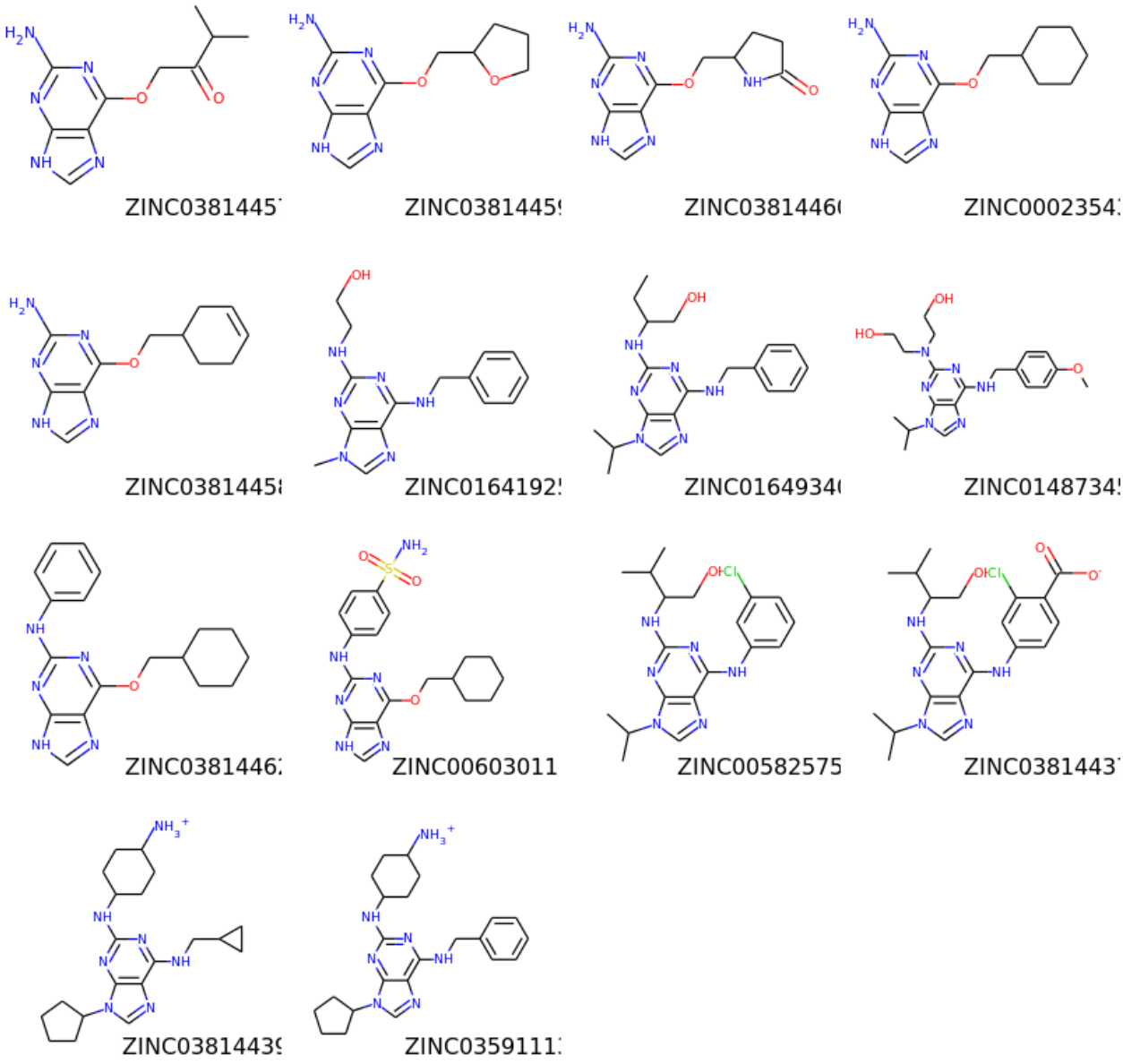
The result looks like this:



These would of course look better if the common core were aligned. This is easy enough to do:

```
>>> p = Chem.MolFromSmiles('[nH]1cnc2cncnc21')
>>> subms = [x for x in ms if x.HasSubstructMatch(p)]
>>> len(subms)
14
>>> AllChem.Compute2DCoords(p)
0
>>> for m in subms: AllChem.GenerateDepictionMatchingDStructure(m,p)
>>> img=Draw.MolsToGridImage(subms,molsPerRow=4,subImgSize=(200,200),legends=[x.GetProp("_Name") for
>>> img.save('images/cdk2_molgrid.aligned.png')
```

The result looks like this:



3.4 Substructure Searching

Substructure matching can be done using query molecules built from SMARTS:

```
>>> m = Chem.MolFromSmiles('c1ccccc1O')
>>> patt = Chem.MolFromSmarts('ccO')
>>> m.HasSubstructMatch(patt)
True
>>> m.GetSubstructMatch(patt)
(0, 5, 6)
```

Those are the atom indices in `m`, ordered as `patt`'s atoms. To get all of the matches:

```
>>> m.GetSubstructMatches(patt)
((0, 5, 6), (4, 5, 6))
```

This can be used to easily filter lists of molecules:

```
>>> suppl = Chem.SDMolSupplier('data/actives_5ht3.sdf')
>>> patt = Chem.MolFromSmarts('c[NH1]')
>>> matches = []
>>> for mol in suppl:
...   if mol.HasSubstructMatch(patt):
...     matches.append(mol)
...
>>> len(matches)
22
```

We can write the same thing more compactly using Python's list comprehension syntax:

```
>>> matches = [x for x in suppl if x.HasSubstructMatch(patt)]
>>> len(matches)
22
```

Substructure matching can also be done using molecules built from SMILES instead of SMARTS:

```
>>> m = Chem.MolFromSmiles('C1=CC=CC=C1OC')
>>> m.HasSubstructMatch(Chem.MolFromSmarts('CO'))
True
>>> m.HasSubstructMatch(Chem.MolFromSmiles('CO'))
True
```

But don't forget that the semantics of the two languages are not exactly equivalent:

```
>>> m.HasSubstructMatch(Chem.MolFromSmiles('COC'))
True
>>> m.HasSubstructMatch(Chem.MolFromSmarts('COC'))
False
>>> m.HasSubstructMatch(Chem.MolFromSmarts('COc')) #<- need an aromatic C
True
```

3.5 Chemical Transformations

The RDKit contains a number of functions for modifying molecules. Note that these transformation functions are intended to provide an easy way to make simple modifications to molecules. For more complex transformations, use the Chemical Reactions functionality.

3.5.1 Substructure-based transformations

There's a variety of functionality for using the RDKit's substructure-matching machinery for doing quick molecular transformations. These transformations include deleting substructures:

```
>>> m = Chem.MolFromSmiles('CC(=O)O')
>>> patt = Chem.MolFromSmarts('C(=O)[OH]')
>>> rm = AllChem.DeleteSubstructs(m,patt)
>>> Chem.MolToSmiles(rm)
'C'
```

replacing substructures:

```
>>> repl = Chem.MolFromSmiles('OC')
>>> patt = Chem.MolFromSmarts('[$(NC(=O))]')
>>> m = Chem.MolFromSmiles('CC(=O)N')
>>> rms = AllChem.ReplaceSubstructs(m,patt,repl)
>>> rms
(<rdkit.Chem.rdchem.Mol object at 0x...>,)
>>> Chem.MolToSmiles(rms[0])
'COC(C)=O'
```

as well as simple SAR-table transformations like removing side chains:

```
>>> m1 = Chem.MolFromSmiles('BrCCc1cncnc1C(=O)O')
>>> core = Chem.MolFromSmiles('c1cncnc1')
>>> tmp = Chem.ReplaceSidechains(m1,core)
>>> Chem.MolToSmiles(tmp)
'[*]c1cncnc1[*]'
```

and removing cores:

```
>>> tmp = Chem.ReplaceCore(m1,core)
>>> Chem.MolToSmiles(tmp)
'[*]CCBr.[*]C(=O)O'
```

To get more detail about the sidechains (e.g. sidechain labels), use isomeric smiles:

```
>>> Chem.MolToSmiles(tmp,True)
'[1*]CCBr.[2*]C(=O)O'
```

By default the sidechains are labeled based on the order they are found. They can also be labeled according by the number of that core-atom they're attached to:

```
>>> m1 = Chem.MolFromSmiles('c1c(CCO)ncnc1C(=O)O')
>>> tmp=Chem.ReplaceCore(m1,core,labelByIndex=True)
>>> Chem.MolToSmiles(tmp,True)
'[1*]CCO.[5*]C(=O)O'
```

`rdkit.Chem.rdmolops.ReplaceCore` returns the sidechains in a single molecule. This can be split into separate molecules using `rdkit.Chem.rdmolops.GetMolFrags`:

```
>>> rs = Chem.GetMolFrags(tmp,asMols=True)
>>> len(rs)
2
>>> Chem.MolToSmiles(rs[0],True)
'[1*]CCO'
>>> Chem.MolToSmiles(rs[1],True)
'[5*]C(=O)O'
```

3.5.2 Murcko Decomposition

The RDKit provides standard Murcko-type decomposition [3] of molecules into scaffolds:

```
>>> from rdkit.Chem.Scaffolds import MurckoScaffold
>>> cdk2mols = Chem.SDMolSupplier('data/cdk2.sdf')
>>> m1 = cdk2mols[0]
>>> core = MurckoScaffold.GetScaffoldForMol(m1)
>>> Chem.MolToSmiles(core)
'c1nc2cncnc2[nH]1'
```

[3] Bemis, G. W.; Murcko, M. A. "The Properties of Known Drugs. 1. Molecular Frameworks." *J. Med. Chem.* **39**:2887-93 (1996).

or into a generic framework:

```
>>> fw = MurckoScaffold.MakeScaffoldGeneric(core)
>>> Chem.MolToSmiles(fw)
'C1CC2CCCCC2C1'
```

3.6 Maximum Common Substructure

The FindMCS function find a maximum common substructure (MCS) of two or more molecules:

```
>>> from rdkit.Chem import MCS
>>> mol1 = Chem.MolFromSmiles("O=C(NCc1cc(OC)c(O)cc1)CCCC/C=C/C(C)C")
>>> mol2 = Chem.MolFromSmiles("CC(C)CCCCCC(=O)NCC1=CC(=C(C=C1)O)OC")
>>> mol3 = Chem.MolFromSmiles("c1(C=O)cc(OC)c(O)cc1")
>>> mols = [mol1,mol2,mol3]
>>> MCS.FindMCS(mols)
MCSResult(numAtoms=10, numBonds=10, smarts='[#6]-[#6]:1:[#6]:[#6](:[#6](:[#6]:[#6]:1)-[#8])-[#8]-[#6]
```

It returns an MCSResult instance with information about the number of atoms and bonds in the MCS, the SMARTS string which matches the identified MCS, and a flag saying if the algorithm timed out. If no MCS is found then the number of atoms and bonds is set to -1 and the SMARTS to `None`. This can be because the MCS is smaller than `minNumAtoms`. Normally this is two atoms, but you can specify a higher value.

By default, two atoms match if they are the same element and two bonds match if they have the same bond type. Specify `atomCompare` and `bondCompare` to use different comparison functions, as in:

```
>>> mols = (Chem.MolFromSmiles('NCC'),Chem.MolFromSmiles('OC=C'))
>>> MCS.FindMCS(mols)
MCSResult(numAtoms=-1, numBonds=-1, smarts=None, completed=1)
>>> MCS.FindMCS(mols, atomCompare="any")
MCSResult(numAtoms=2, numBonds=1, smarts='[*]-[*]', completed=1)
>>> MCS.FindMCS(mols, bondCompare="any")
MCSResult(numAtoms=2, numBonds=1, smarts='[#6]~[#6]', completed=1)
```

An atomCompare of "any" says that any atom matches any other atom, "elements" compares by element type, and "isotopes" matches based on the isotope label. Isotope labels can be used to implement user-defined atom types. A bondCompare of "any" says that any bond matches any other bond, and "bondtypes" says bonds are equivalent if and only if they have the same bond type.

A substructure has both atoms and bonds. The default `maximize` setting of "atoms" finds a common substructure with the most number of atoms. Use maximize="bonds" to maximize the number of bonds. Maximizing the number of bonds tends to maximize the number of rings, although two small rings may have fewer bonds than one large ring.

You might not want a 3-valent nitrogen to match one which is 5-valent. The default `matchValences` value of False ignores valence information. When True, the atomCompare setting is modified to also require that the two atoms have the same valency.

```
>>> mols = (Chem.MolFromSmiles('NC1OC1'),Chem.MolFromSmiles('C1OC1[N+](=O)[O-]'))
>>> MCS.FindMCS(mols)
MCSResult(numAtoms=4, numBonds=4, smarts='[#7]-[#6]-1-[#8]-[#6]-1', completed=1)
>>> MCS.FindMCS(mols, matchValences=True)
MCSResult(numAtoms=3, numBonds=3, smarts='[#6v4]-1-[#8v2]-[#6v4]-1', completed=1)
```

It can be strange to see a linear carbon chain match a carbon ring, which is what the `ringMatchesRingOnly` default of False does. If you set it to True then ring bonds will only match ring bonds.

```
>>> mols = [Chem.MolFromSmiles("C1CCC1CCC"), Chem.MolFromSmiles("C1CCCCCC1")]
>>> MCS.FindMCS(mols)
MCSResult(numAtoms=7, numBonds=6, smarts='[#6]-[#6]-[#6]-[#6]-[#6]-[#6]-[#6]', completed=1)
>>> MCS.FindMCS(mols, ringMatchesRingOnly=True)
MCSResult(numAtoms=4, numBonds=3, smarts='[#6](-@[#6])-@[#6]-@[#6]', completed=1)
```

You can further restrict things and require that partial rings (as in this case) are not allowed. That is, if an atom is part of the MCS and the atom is in a ring of the entire molecule then that atom is also in a ring of the MCS. Set `completeRingsOnly` to True to toggle this requirement and also sets ringMatchesRingOnly to True.

```
>>> mols = [Chem.MolFromSmiles("CCC1CC2C1CN2"), Chem.MolFromSmiles("C1CC2C1CC2")]
>>> MCS.FindMCS(mols)
MCSResult(numAtoms=6, numBonds=6, smarts='[#6]-1-[#6]-[#6](-[#6])-[#6]-1-[#6]', completed=1)
>>> MCS.FindMCS(mols, ringMatchesRingOnly=True)
MCSResult(numAtoms=5, numBonds=5, smarts='[#6]-@1-@[#6]-@[#6]-@[#6]-@1-@[#6]', completed=1)
>>> MCS.FindMCS(mols, completeRingsOnly=True)
MCSResult(numAtoms=4, numBonds=4, smarts='[#6]-@1-@[#6]-@[#6]-@[#6]-@1', completed=1)
```

The MCS algorithm will exhaustively search for a maximum common substructure. Typically this takes a fraction of a second, but for some comparisons this can take minutes or longer. Use the `timeout` parameter to stop the search after the given number of seconds (wall-clock seconds, not CPU seconds) and return the best match found in that time. If timeout is reached then the `completed` property of the MCSResult will be 0 instead of 1.

```
>>> mols = [Chem.MolFromSmiles("Nc1ccccc1"*100), Chem.MolFromSmiles("Nc1ccccccccc1"*100)]
>>> MCS.FindMCS(mols, timeout=0.1)
MCSResult(numAtoms=..., numBonds=..., smarts='[#7]-[#6]...', completed=0)
```

(The MCS after 50 seconds contained 511 atoms.)

3.7 Fingerprinting and Molecular Similarity

The RDKit has a variety of built-in functionality for generating molecular fingerprints and using them to calculate molecular similarity.

3.7.1 Topological Fingerprints

```
>>> from rdkit import DataStructs
>>> from rdkit.Chem.Fingerprints import FingerprintMols
>>> ms = [Chem.MolFromSmiles('CCOC'), Chem.MolFromSmiles('CCO'),
... Chem.MolFromSmiles('COC')]
>>> fps = [FingerprintMols.FingerprintMol(x) for x in ms]
>>> DataStructs.FingerprintSimilarity(fps[0],fps[1])
0.666...
>>> DataStructs.FingerprintSimilarity(fps[0],fps[2])
0.444...
>>> DataStructs.FingerprintSimilarity(fps[1],fps[2])
0.25
```

The fingerprinting algorithm used is similar to that used in the Daylight fingerprinter: it identifies and hashes topological paths (e.g. along bonds) in the molecule and then uses them to set bits in a fingerprint of user-specified lengths. After all paths have been identified, the fingerprint is typically folded down until a particular density of set bits is obtained.

The default set of parameters used by the fingerprinter is: - minimum path size: 1 bond - maximum path size: 7 bonds - fingerprint size: 2048 bits - number of bits set per hash: 2 - minimum fingerprint size: 64 bits - target on-bit density

0.3

You can control these by calling `rdkit.Chem.rdmolops.RDKFingerprint` directly; this will return an unfolded fingerprint that you can then fold to the desired density. The function `rdkit.Chem.Fingerprints.FingerprintMols.FingerprintMol` (written in python) shows how this is done.

The default similarity metric used by `rdkit.DataStructs.FingerprintSimilarity` is the Tanimoto similarity. One can use different similarity metrics:

```
>>> DataStructs.FingerprintSimilarity(fps[0],fps[1], metric=DataStructs.DiceSimilarity)
0.8
```

Available similarity metrics include Tanimoto, Dice, Cosine, Sokal, Russel, Kulczynski, McConnaughey, and Tversky.

3.7.2 MACCS Keys

There is a SMARTS-based implementation of the 166 public MACCS keys.

```
>>> from rdkit.Chem import MACCSkeys
>>> fps = [MACCSkeys.GenMACCSKeys(x) for x in ms]
>>> DataStructs.FingerprintSimilarity(fps[0],fps[1])
0.5
>>> DataStructs.FingerprintSimilarity(fps[0],fps[2])
0.538...
>>> DataStructs.FingerprintSimilarity(fps[1],fps[2])
0.214...
```

The MACCS keys were critically evaluated and compared to other MACCS implementations in Q3 2008. In cases where the public keys are fully defined, things looked pretty good.

3.7.3 Atom Pairs and Topological Torsions

Atom-pair descriptors [4] are available in several different forms. The standard form is as fingerprint including counts for each bit instead of just zeros and ones:

```
>>> from rdkit.Chem.AtomPairs import Pairs
>>> ms = [Chem.MolFromSmiles('C1CCC1OCC'),Chem.MolFromSmiles('CC(C)OCC'),Chem.MolFromSmiles('CCOCC')]
>>> pairFps = [Pairs.GetAtomPairFingerprint(x) for x in ms]
```

Because the space of bits that can be included in atom-pair fingerprints is huge, they are stored in a sparse manner. We can get the list of bits and their counts for each fingerprint as a dictionary:

```
>>> d = pairFps[-1].GetNonzeroElements()
>>> d[541732]
1
>>> d[1606690]
2
```

Descriptions of the bits are also available:

```
>>> Pairs.ExplainPairScore(558115)
(('C', 1, 0), 3, ('C', 2, 0))
```

[4] Carhart, R.E.; Smith, D.H.; Venkataraghavan R. "Atom Pairs as Molecular Features in Structure-Activity Studies: Definition and Applications" *J. Chem. Inf. Comp. Sci.* **25**:64-73 (1985).

The above means: C with 1 neighbor and 0 pi electrons which is 3 bonds from a C with 2 neighbors and 0 pi electrons

The usual metric for similarity between atom-pair fingerprints is Dice similarity:

```
>>> from rdkit import DataStructs
>>> DataStructs.DiceSimilarity(pairFps[0],pairFps[1])
0.333...
>>> DataStructs.DiceSimilarity(pairFps[0],pairFps[2])
0.258...
>>> DataStructs.DiceSimilarity(pairFps[1],pairFps[2])
0.56
```

It's also possible to get atom-pair descriptors encoded as a standard bit vector fingerprint (ignoring the count information):

```
>>> pairFps = [Pairs.GetAtomPairFingerprintAsBitVect(x) for x in ms]
```

Since these are standard bit vectors, the `rdkit.DataStructs` module can be used for similarity:

```
>>> from rdkit import DataStructs
>>> DataStructs.DiceSimilarity(pairFps[0],pairFps[1])
0.48
>>> DataStructs.DiceSimilarity(pairFps[0],pairFps[2])
0.380...
>>> DataStructs.DiceSimilarity(pairFps[1],pairFps[2])
0.625
```

Topological torsion descriptors [5] are calculated in essentially the same way:

```
>>> from rdkit.Chem.AtomPairs import Torsions
>>> tts = [Torsions.GetTopologicalTorsionFingerprintAsIntVect(x) for x in ms]
>>> DataStructs.DiceSimilarity(tts[0],tts[1])
0.166...
```

At the time of this writing, topological torsion fingerprints have too many bits to be encodeable using the BitVector machinery, so there is no GetTopologicalTorsionFingerprintAsBitVect function.

3.7.4 Morgan Fingerprints (Circular Fingerprints)

This family of fingerprints, better known as circular fingerprints [6], is built by applying the Morgan algorithm to a set of user-supplied atom invariants. When generating Morgan fingerprints, the radius of the fingerprint must also be provided :

```
>>> from rdkit.Chem import AllChem
>>> m1 = Chem.MolFromSmiles('Cc1ccccc1')
>>> fp1 = AllChem.GetMorganFingerprint(m1,2)
>>> fp1
<rdkit.DataStructs.cDataStructs.UIntSparseIntVect object at 0x...>
>>> m2 = Chem.MolFromSmiles('Cc1ncccc1')
>>> fp2 = AllChem.GetMorganFingerprint(m2,2)
>>> DataStructs.DiceSimilarity(fp1,fp2)
0.55...
```

Morgan fingerprints, like atom pairs and topological torsions, use counts by default, but it's also possible to calculate them as bit vectors:

[5] Nilakantan, R.; Bauman N.; Dixon J.S.; Venkataraghavan R. "Topological Torsion: A New Molecular Descriptor for SAR Applications. Comparison with Other Desciptors." *J. Chem.Inf. Comp. Sci.* **27**:82-5 (1987).

[6] Rogers, D.; Hahn, M. "Extended-Connectivity Fingerprints." *J. Chem. Inf. and Model.* **50**:742-54 (2010).

```
>>> fp1 = AllChem.GetMorganFingerprintAsBitVect(m1,2,nBits=1024)
>>> fp1
<rdkit.DataStructs.cDataStructs.ExplicitBitVect object at 0x...>
>>> fp2 = AllChem.GetMorganFingerprintAsBitVect(m2,2,nBits=1024)
>>> DataStructs.DiceSimilarity(fp1,fp2)
0.51...
```

The default atom invariants use connectivity information similar to those used for the well known ECFP family of fingerprints. Feature-based invariants, similar to those used for the FCFP fingerprints, can also be used. The feature definitions used are defined in the section Feature Definitions Used in the Morgan Fingerprints. At times this can lead to quite different similarity scores:

```
>>> m1 = Chem.MolFromSmiles('c1ccccn1')
>>> m2 = Chem.MolFromSmiles('c1ccco1')
>>> fp1 = AllChem.GetMorganFingerprint(m1,2)
>>> fp2 = AllChem.GetMorganFingerprint(m2,2)
>>> ffp1 = AllChem.GetMorganFingerprint(m1,2,useFeatures=True)
>>> ffp2 = AllChem.GetMorganFingerprint(m2,2,useFeatures=True)
>>> DataStructs.DiceSimilarity(fp1,fp2)
0.36...
>>> DataStructs.DiceSimilarity(ffp1,ffp2)
0.90...
```

When comparing the ECFP/FCFP fingerprints and the Morgan fingerprints generated by the RDKit, remember that the 4 in ECFP4 corresponds to the diameter of the atom environments considered, while the Morgan fingerprints take a radius parameter. So the examples above, with radius=2, are roughly equivalent to ECFP4 and FCFP4.

The user can also provide their own atom invariants using the optional invariants argument to `rdkit.Chem.rdMolDescriptors.GetMorganFingerprint`. Here's a simple example that uses a constant for the invariant; the resulting fingerprints compare the topology of molecules:

```
>>> m1 = Chem.MolFromSmiles('Cc1ccccc1')
>>> m2 = Chem.MolFromSmiles('Cc1ncncn1')
>>> fp1 = AllChem.GetMorganFingerprint(m1,2,invariants=[1]*m1.GetNumAtoms())
>>> fp2 = AllChem.GetMorganFingerprint(m2,2,invariants=[1]*m2.GetNumAtoms())
>>> fp1==fp2
True
```

Note that bond order is by default still considered:

```
>>> m3 = Chem.MolFromSmiles('CC1CCCCC1')
>>> fp3 = AllChem.GetMorganFingerprint(m3,2,invariants=[1]*m3.GetNumAtoms())
>>> fp1==fp3
False
```

But this can also be turned off:

```
>>> fp1 = AllChem.GetMorganFingerprint(m1,2,invariants=[1]*m1.GetNumAtoms(),
... useBondTypes=False)
>>> fp3 = AllChem.GetMorganFingerprint(m3,2,invariants=[1]*m3.GetNumAtoms(),
... useBondTypes=False)
>>> fp1==fp3
True
```

Explaining bits from Morgan Fingerprints

Information is available about the atoms that contribute to particular bits in the Morgan fingerprint via the bitInfo argument. The dictionary provided is populated with one entry per bit set in the fingerprint, the keys are the bit ids, the values are lists of (atom index, radius) tuples.

```
>>> m = Chem.MolFromSmiles('c1cccnc1C')
>>> info={}
>>> fp = AllChem.GetMorganFingerprint(m,2,bitInfo=info)
>>> len(fp.GetNonzeroElements())
16
>>> len(info)
16
>>> info[98513984]
((1, 1), (2, 1))
>>> info[4048591891]
((5, 2),)
```

Interpreting the above: bit 98513984 is set twice: once by atom 1 and once by atom 2, each at radius 1. Bit 4048591891 is set once by atom 5 at radius 2.

Focusing on bit 4048591891, we can extract the submolecule consisting of all atoms within a radius of 2 of atom 5:

```
>>> env = Chem.FindAtomEnvironmentOfRadiusN(m,2,5)
>>> amap={}
>>> submol=Chem.PathToSubmol(m,env,atomMap=amap)
>>> submol.GetNumAtoms()
6
>>> amap
{0: 3, 1: 5, 3: 4, 4: 0, 5: 1, 6: 2}
```

And then "explain" the bit by generating SMILES for that submolecule:

```
>>> Chem.MolToSmiles(submol)
'ccc(C)nc'
```

This is more useful when the SMILES is rooted at the central atom:

```
>>> Chem.MolToSmiles(submol,rootedAtAtom=amap[5],canonical=False)
'c(nc)(C)cc'
```

An alternate (and faster, particularly for large numbers of molecules) approach to do the same thing, using the function `rdkit.Chem.MolFragmentToSmiles`:

```
>>> atoms=set()
>>> for bidx in env:
...     atoms.add(m.GetBondWithIdx(bidx).GetBeginAtomIdx())
...     atoms.add(m.GetBondWithIdx(bidx).GetEndAtomIdx())
...
>>> Chem.MolFragmentToSmiles(m,atomsToUse=list(atoms),bondsToUse=env,rootedAtAtom=5)
'c(C)(cc)nc'
```

3.7.5 Picking Diverse Molecules Using Fingerprints

A common task is to pick a small subset of diverse molecules from a larger set. The RDKit provides a number of approaches for doing this in the `rdkit.SimDivFilters` module. The most efficient of these uses the MaxMin algorithm. [7] Here's an example:

Start by reading in a set of molecules and generating Morgan fingerprints:

```
>>> from rdkit import Chem
>>> from rdkit.Chem.rdMolDescriptors import GetMorganFingerprint
```

[7] Ashton, M. et al. "Identification of Diverse Database Subsets using Property-Based and Fragment-Based Molecular Descriptions." *Quantitative Structure-Activity Relationships* **21**:598-604 (2002).

```
>>> from rdkit import DataStructs
>>> from rdkit.SimDivFilters.rdSimDivPickers import MaxMinPicker
>>> ms = [x for x in Chem.SDMolSupplier('data/actives_5ht3.sdf')]
>>> while ms.count(None): ms.remove(None)
>>> fps = [GetMorganFingerprint(x,3) for x in ms]
>>> nfps = len(fps)
```

The algorithm requires a function to calculate distances between objects, we'll do that using DiceSimilarity:

```
>>> def distij(i,j,fps=fps):
...   return 1-DataStructs.DiceSimilarity(fps[i],fps[j])
```

Now create a picker and grab a set of 10 diverse molecules:

```
>>> picker = MaxMinPicker()
>>> pickIndices = picker.LazyPick(distij,nfps,10,seed=23)
>>> list(pickIndices)
[93, 109, 154, 6, 95, 135, 151, 61, 137, 139]
```

Note that the picker just returns indices of the fingerprints; we can get the molecules themselves as follows:

```
>>> picks = [ms[x] for x in pickIndices]
```

3.8 Descriptor Calculation

A variety of descriptors are available within the RDKit. The complete list is provided in List of Available Descriptors.

Most of the descriptors are straightforward to use from Python via the centralized `rdkit.Chem.Descriptors` module :

```
>>> from rdkit.Chem import Descriptors
>>> m = Chem.MolFromSmiles('c1ccccc1C(=O)O')
>>> Descriptors.TPSA(m)
37.3
>>> Descriptors.MolLogP(m)
1.3848
```

Partial charges are handled a bit differently:

```
>>> m = Chem.MolFromSmiles('c1ccccc1C(=O)O')
>>> AllChem.ComputeGasteigerCharges(m)
>>> float(m.GetAtomWithIdx(0).GetProp('_GasteigerCharge'))
-0.047...
```

3.9 Chemical Reactions

The RDKit also supports applying chemical reactions to sets of molecules. One way of constructing chemical reactions is to use a SMARTS-based language similar to Daylight's Reaction SMILES [8]:

```
>>> rxn = AllChem.ReactionFromSmarts('[C:1](=[O:2])-[OD1].[N!H0:3]>>[C:1](=[O:2])[N:3]')
>>> rxn
<rdkit.Chem.rdChemReactions.ChemicalReaction object at 0x...>
>>> rxn.GetNumProductTemplates()
1
```

[8] A more detailed description of reaction smarts, as defined by the rdkit, is in the *The RDKit Book*.

```
>>> ps = rxn.RunReactants((Chem.MolFromSmiles('CC(=O)O'),Chem.MolFromSmiles('NC')))
>>> len(ps) # one entry for each possible set of products
1
>>> len(ps[0]) # each entry contains one molecule for each product
1
>>> Chem.MolToSmiles(ps[0][0])
'CNC(C)=O'
>>> ps = rxn.RunReactants((Chem.MolFromSmiles('C(COC(=O)O)C(=O)O'),Chem.MolFromSmiles('NC')))
>>> len(ps)
2
>>> Chem.MolToSmiles(ps[0][0])
'CNC(=O)OCCC(=O)O'
>>> Chem.MolToSmiles(ps[1][0])
'CNC(=O)CCOC(=O)O'
```

Reactions can also be built from MDL rxn files:

```
>>> rxn = AllChem.ReactionFromRxnFile('data/AmideBond.rxn')
>>> rxn.GetNumReactantTemplates()
2
>>> rxn.GetNumProductTemplates()
1
>>> ps = rxn.RunReactants((Chem.MolFromSmiles('CC(=O)O'), Chem.MolFromSmiles('NC')))
>>> len(ps)
1
>>> Chem.MolToSmiles(ps[0][0])
'CNC(C)=O'
```

It is, of course, possible to do reactions more complex than amide bond formation:

```
>>> rxn = AllChem.ReactionFromSmarts('[C:1]=[C:2].[C:3]=[*:4][*:5]=[C:6]>>[C:1]1[C:2][C:3][*:4]=[*:5]
>>> ps = rxn.RunReactants((Chem.MolFromSmiles('OC=C'), Chem.MolFromSmiles('C=CC(N)=C')))
>>> Chem.MolToSmiles(ps[0][0])
'NC1=CCCC(O)C1'
```

Note in this case that there are multiple mappings of the reactants onto the templates, so we have multiple product sets:

```
>>> len(ps)
4
```

You can use canonical smiles and a python dictionary to get the unique products:

```
>>> uniqps = {}
>>> for p in ps:
...   smi = Chem.MolToSmiles(p[0])
...   uniqps[smi] = p[0]
...
>>> uniqps.keys()
['NC1=CCC(O)CC1', 'NC1=CCCC(O)C1']
```

Note that the molecules that are produced by the chemical reaction processing code are not sanitized, as this artificial reaction demonstrates:

```
>>> rxn = AllChem.ReactionFromSmarts('[C:1]=[C:2][C:3]=[C:4].[C:5]=[C:6]>>[C:1]1=[C:2][C:3]=[C:4][C:5
>>> ps = rxn.RunReactants((Chem.MolFromSmiles('C=CC=C'), Chem.MolFromSmiles('C=C')))
>>> Chem.MolToSmiles(ps[0][0])
'C1=CC=CC=C1'
>>> p0 = ps[0][0]
>>> Chem.SanitizeMol(p0)
```

```
rdkit.Chem.rdmolops.SanitizeFlags.SANITIZE_NONE
>>> Chem.MolToSmiles(p0)
'c1ccccc1'
```

3.9.1 Advanced Reaction Functionality

Protecting Atoms

Sometimes, particularly when working with rxn files, it is difficult to express a reaction exactly enough to not end up with extraneous products. The RDKit provides a method of "protecting" atoms to disallow them from taking part in reactions.

This can be demonstrated re-using the amide-bond formation reaction used above. The query for amines isn't specific enough, so it matches any nitrogen that has at least one H attached. So if we apply the reaction to a molecule that already has an amide bond, the amide N is also treated as a reaction site:

```
>>> rxn = AllChem.ReactionFromRxnFile('data/AmideBond.rxn')
>>> acid = Chem.MolFromSmiles('CC(=O)O')
>>> base = Chem.MolFromSmiles('CC(=O)NCCN')
>>> ps = rxn.RunReactants((acid,base))
>>> len(ps)
2
>>> Chem.MolToSmiles(ps[0][0])
'CC(=O)N(CCN)C(C)=O'
>>> Chem.MolToSmiles(ps[1][0])
'CC(=O)NCCNC(C)=O'
```

The first product corresponds to the reaction at the amide N.

We can prevent this from happening by protecting all amide Ns. Here we do it with a substructure query that matches amides and thioamides and then set the "_protected" property on matching atoms:

```
>>> amidep = Chem.MolFromSmarts('[N;$(NC=[O,S])]')
>>> for match in base.GetSubstructMatches(amidep):
...     base.GetAtomWithIdx(match[0]).SetProp('_protected','1')
```

Now the reaction only generates a single product:

```
>>> ps = rxn.RunReactants((acid,base))
>>> len(ps)
1
>>> Chem.MolToSmiles(ps[0][0])
'CC(=O)NCCNC(C)=O'
```

3.9.2 Recap Implementation

Associated with the chemical reaction functionality is an implementation of the Recap algorithm. [9] Recap uses a set of chemical transformations mimicking common reactions carried out in the lab in order to decompose a molecule into a series of reasonable fragments.

The RDKit `rdkit.Chem.Recap` implementation keeps track of the hierarchy of transformations that were applied:

[9] Lewell, X.Q.; Judd, D.B.; Watson, S.P.; Hann, M.M. "RECAP-Retrosynthetic Combinatorial Analysis Procedure: A Powerful New Technique for Identifying Privileged Molecular Fragments with Useful Applications in Combinatorial Chemistry" *J. Chem. Inf. Comp. Sci.* **38**:511-22 (1998).

```
>>> from rdkit import Chem
>>> from rdkit.Chem import Recap
>>> m = Chem.MolFromSmiles('c1ccccc1OCCOC(=O)CC')
>>> hierarch = Recap.RecapDecompose(m)
>>> type(hierarch)
<class 'rdkit.Chem.Recap.RecapHierarchyNode'>
```

The hierarchy is rooted at the original molecule:

```
>>> hierarch.smiles
'CCC(=O)OCCOc1ccccc1'
```

and each node tracks its children using a dictionary keyed by SMILES:

```
>>> ks=hierarch.children.keys()
>>> ks.sort()
>>> ks
['[*]C(=O)CC', '[*]CCOC(=O)CC', '[*]CCOc1ccccc1', '[*]OCCOc1ccccc1', '[*]c1ccccc1']
```

The nodes at the bottom of the hierarchy (the leaf nodes) are easily accessible, also as a dictionary keyed by SMILES:

```
>>> ks=hierarch.GetLeaves().keys()
>>> ks.sort()
>>> ks
['[*]C(=O)CC', '[*]CCO[*]', '[*]CCOc1ccccc1', '[*]c1ccccc1']
```

Notice that dummy atoms are used to mark points where the molecule was fragmented.

The nodes themselves have associated molecules:

```
>>> leaf = hierarch.GetLeaves()[ks[0]]
>>> Chem.MolToSmiles(leaf.mol)
'[*]C(=O)CC'
```

3.9.3 BRICS Implementation

The RDKit also provides an implementation of the BRICS algorithm. [10] BRICS provides another method for fragmenting molecules along synthetically accessible bonds:

```
>>> from rdkit.Chem import BRICS
>>> cdk2mols = Chem.SDMolSupplier('data/cdk2.sdf')
>>> m1 = cdk2mols[0]
>>> list(BRICS.BRICSDecompose(m1))
['[4*]CC(=O)C(C)C', '[14*]c1nc(N)nc2[nH]cnc21', '[3*]O[3*]']
>>> m2 = cdk2mols[20]
>>> list(BRICS.BRICSDecompose(m2))
['[3*]OC', '[1*]C(=O)NN(C)C', '[14*]c1[nH]nc2c1C(=O)c1c-2cccc1[16*]', '[5*]N[5*]', '[16*]c1ccc([16*])
```

Notice that RDKit BRICS implementation returns the unique fragments generated from a molecule and that the dummy atoms are tagged to indicate which type of reaction applies.

It's quite easy to generate the list of all fragments for a group of molecules:

```
>>> allfrags=set()
>>> for m in cdk2mols:
...     pieces = BRICS.BRICSDecompose(m)
```

[10] Degen, J.; Wegscheid-Gerlach, C.; Zaliani, A; Rarey, M. "On the Art of Compiling and Using 'Drug-Like' Chemical Fragment Spaces." *ChemMedChem* **3**:1503–7 (2008).

```
...     allfrags.update(pieces)
>>> len(allfrags)
90
>>> list(allfrags)[:5]
['[4*]CC[NH3+]', '[14*]c1cnc[nH]1', '[16*]c1cc([16*])c2c3c(ccc2F)NC(=O)c31', '[16*]c1ccc([16*])c(Cl)c
```

The BRICS module also provides an option to apply the BRICS rules to a set of fragments to create new molecules:

```
>>> import random
>>> random.seed(127)
>>> fragms = [Chem.MolFromSmiles(x) for x in allfrags]
>>> ms = BRICS.BRICSBuild(fragms)
```

The result is a generator object:

```
>>> ms
<generator object BRICSBuild at 0x...>
```

That returns molecules on request:

```
>>> prods = [ms.next() for x in range(10)]
>>> Chem.MolToSmiles(prods[0],True)
'O=[N+]([O-])c1ccc(C2CCCO2)cc1'
>>> Chem.MolToSmiles(prods[1],True)
'c1ccc(C2CCCO2)cc1'
>>> Chem.MolToSmiles(prods[2],True)
'NS(=O)(=O)c1ccc(C2CCCO2)cc1'
```

3.10 Chemical Features and Pharmacophores

3.10.1 Chemical Features

Chemical features in the RDKit are defined using a SMARTS-based feature definition language (described in detail in the RDKit book). To identify chemical features in molecules, you first must build a feature factory:

```
>>> from rdkit import Chem
>>> from rdkit.Chem import ChemicalFeatures
>>> from rdkit import RDConfig
>>> import os
>>> fdefName = os.path.join(RDConfig.RDDataDir,'BaseFeatures.fdef')
>>> factory = ChemicalFeatures.BuildFeatureFactory(fdefName)
```

and then use the factory to search for features:

```
>>> m = Chem.MolFromSmiles('OCc1ccccc1CN')
>>> feats = factory.GetFeaturesForMol(m)
>>> len(feats)
8
```

The individual features carry information about their family (e.g. donor, acceptor, etc.), type (a more detailed description), and the atom(s) that is/are associated with the feature:

```
>>> feats[0].GetFamily()
'Donor'
>>> feats[0].GetType()
'SingleAtomDonor'
>>> feats[0].GetAtomIds()
```

```
(0,)
>>> feats[4].GetFamily()
'Aromatic'
>>> feats[4].GetAtomIds()
(2, 3, 4, 5, 6, 7)
```

If the molecule has coordinates, then the features will also have reasonable locations:

```
>>> from rdkit.Chem import AllChem
>>> AllChem.Compute2DCoords(m)
0
>>> feats[0].GetPos()
<rdkit.Geometry.rdGeometry.Point3D object at 0x...>
>>> list(feats[0].GetPos())
[-2.99..., -1.558..., 0.0]
```

3.10.2 2D Pharmacophore Fingerprints

Combining a set of chemical features with the 2D (topological) distances between them gives a 2D pharmacophore. When the distances are binned, unique integer ids can be assigned to each of these pharmacophores and they can be stored in a fingerprint. Details of the encoding are in the *The RDKit Book*.

Generating pharmacophore fingerprints requires chemical features generated via the usual RDKit feature-typing mechanism:

```
>>> from rdkit import Chem
>>> from rdkit.Chem import ChemicalFeatures
>>> fdefName = 'data/MinimalFeatures.fdef'
>>> featFactory = ChemicalFeatures.BuildFeatureFactory(fdefName)
```

The fingerprints themselves are calculated using a signature (fingerprint) factory, which keeps track of all the parameters required to generate the pharmacophore:

```
>>> from rdkit.Chem.Pharm2D.SigFactory import SigFactory
>>> sigFactory = SigFactory(featFactory,minPointCount=2,maxPointCount=3)
>>> sigFactory.SetBins([(0,2),(2,5),(5,8)])
>>> sigFactory.Init()
>>> sigFactory.GetSigSize()
885
```

The signature factory is now ready to be used to generate fingerprints, a task which is done using the `rdkit.Chem.Pharm2D.Generate` module:

```
>>> from rdkit.Chem.Pharm2D import Generate
>>> mol = Chem.MolFromSmiles('OCC(=O)CCCN')
>>> fp = Generate.Gen2DFingerprint(mol,sigFactory)
>>> fp
<rdkit.DataStructs.cDataStructs.SparseBitVect object at 0x...>
>>> len(fp)
885
>>> fp.GetNumOnBits()
57
```

Details about the bits themselves, including the features that are involved and the binned distance matrix between the features, can be obtained from the signature factory:

```
>>> list(fp.GetOnBits())[:5]
[1, 2, 6, 7, 8]
```

```
>>> sigFactory.GetBitDescription(1)
'Acceptor Acceptor |0 1|1 0|'
>>> sigFactory.GetBitDescription(2)
'Acceptor Acceptor |0 2|2 0|'
>>> sigFactory.GetBitDescription(8)
'Acceptor Donor |0 2|2 0|'
>>> list(fp.GetOnBits())[-5:]
[704, 706, 707, 708, 714]
>>> sigFactory.GetBitDescription(707)
'Donor Donor PosIonizable |0 1 2|1 0 1|2 1 0|'
>>> sigFactory.GetBitDescription(714)
'Donor Donor PosIonizable |0 2 2|2 0 0|2 0 0|'
```

For the sake of convenience (to save you from having to edit the fdef file every time) it is possible to disable particular feature types within the SigFactory:

```
>>> sigFactory.skipFeats=['PosIonizable']
>>> sigFactory.Init()
>>> sigFactory.GetSigSize()
510
>>> fp2 = Generate.Gen2DFingerprint(mol,sigFactory)
>>> fp2.GetNumOnBits()
36
```

Another possible set of feature definitions for 2D pharmacophore fingerprints in the RDKit are those published by Gobbi and Poppinger. [11] The module `rdkit.Chem.Pharm2D.Gobbi_Pharm2D` has a pre-configured signature factory for these fingerprint types. Here's an example of using it:

```
>>> from rdkit import Chem
>>> from rdkit.Chem.Pharm2D import Gobbi_Pharm2D,Generate
>>> m = Chem.MolFromSmiles('OCC=CC(=O)O')
>>> fp = Generate.Gen2DFingerprint(m,Gobbi_Pharm2D.factory)
>>> fp
<rdkit.DataStructs.cDataStructs.SparseBitVect object at 0x...>
>>> fp.GetNumOnBits()
8
>>> list(fp.GetOnBits())
[23, 30, 150, 154, 157, 185, 28878, 30184]
>>> Gobbi_Pharm2D.factory.GetBitDescription(157)
'HA HD |0 3|3 0|'
>>> Gobbi_Pharm2D.factory.GetBitDescription(30184)
'HA HD HD |0 3 0|3 0 3|0 3 0|'
```

3.11 Molecular Fragments

The RDKit contains a collection of tools for fragmenting molecules and working with those fragments. Fragments are defined to be made up of a set of connected atoms that may have associated functional groups. This is more easily demonstrated than explained:

```
>>> fName=os.path.join(RDConfig.RDDataDir,'FunctionalGroups.txt')
>>> from rdkit.Chem import FragmentCatalog
>>> fparams = FragmentCatalog.FragCatParams(1,6,fName)
>>> fparams.GetNumFuncGroups()
39
```

[11] Gobbi, A. & Poppinger, D. "Genetic optimization of combinatorial libraries." *Biotechnology and Bioengineering* **61**:47-54 (1998).

```
>>> fcat=FragmentCatalog.FragCatalog(fparams)
>>> fcgen=FragmentCatalog.FragCatGenerator()
>>> m = Chem.MolFromSmiles('OCC=CC(=O)O')
>>> fcgen.AddFragsFromMol(m,fcat)
3
>>> fcat.GetEntryDescription(0)
'CC<-O>'
>>> fcat.GetEntryDescription(1)
'C<-C(=O)O>=C'
>>> fcat.GetEntryDescription(2)
'C<-C(=O)O>=CC<-O>'
```

The fragments are stored as entries in a `rdkit.Chem.rdfragcatalog.FragCatalog`. Notice that the entry descriptions include pieces in angular brackets (e.g. between '<' and '>'). These describe the functional groups attached to the fragment. For example, in the above example, the catalog entry 0 corresponds to an ethyl fragment with an alcohol attached to one of the carbons and entry 1 is an ethylene with a carboxylic acid on one carbon. Detailed information about the functional groups can be obtained by asking the fragment for the ids of the functional groups it contains and then looking those ids up in the `rdkit.Chem.rdfragcatalog.FragCatParams` object:

```
>>> list(fcat.GetEntryFuncGroupIds(2))
[34, 1]
>>> fparams.GetFuncGroup(1)
<rdkit.Chem.rdchem.Mol object at 0x...>
>>> Chem.MolToSmarts(fparams.GetFuncGroup(1))
'*-C(=O)-,:[O&D1]'
>>> Chem.MolToSmarts(fparams.GetFuncGroup(34))
'*-[O&D1]'
>>> fparams.GetFuncGroup(1).GetProp('_Name')
'-C(=O)O'
>>> fparams.GetFuncGroup(34).GetProp('_Name')
'-O'
```

The catalog is hierarchical: smaller fragments are combined to form larger ones. From a small fragment, one can find the larger fragments to which it contributes using the `rdkit.Chem.rdfragcatalog.FragCatalog.GetEntryDownIds` method:

```
>>> fcat=FragmentCatalog.FragCatalog(fparams)
>>> m = Chem.MolFromSmiles('OCC(NC1CC1)CCC')
>>> fcgen.AddFragsFromMol(m,fcat)
15
>>> fcat.GetEntryDescription(0)
'CC<-O>'
>>> fcat.GetEntryDescription(1)
'CN<-cPropyl>'
>>> list(fcat.GetEntryDownIds(0))
[3, 4]
>>> fcat.GetEntryDescription(3)
'CCC<-O>'
>>> fcat.GetEntryDescription(4)
'C<-O>CN<-cPropyl>'
```

The fragments from multiple molecules can be added to a catalog:

```
>>> suppl = Chem.SmilesMolSupplier('data/bzr.smi')
>>> ms = [x for x in suppl]
>>> fcat=FragmentCatalog.FragCatalog(fparams)
>>> for m in ms: nAdded=fcgen.AddFragsFromMol(m,fcat)
>>> fcat.GetNumEntries()
1169
```

```
>>> fcat.GetEntryDescription(0)
'cC'
>>> fcat.GetEntryDescription(100)
'cc-nc(C)n'
```

The fragments in a catalog are unique, so adding a molecule a second time doesn't add any new entries:

```
>>> fcgen.AddFragsFromMol(ms[0],fcat)
0
>>> fcat.GetNumEntries()
1169
```

Once a `rdkit.Chem.rdfragcatalog.FragCatalog` has been generated, it can be used to fingerprint molecules:

```
>>> fpgen = FragmentCatalog.FragFPGenerator()
>>> fp = fpgen.GetFPForMol(ms[8],fcat)
>>> fp
<rdkit.DataStructs.cDataStructs.ExplicitBitVect object at 0x...>
>>> fp.GetNumOnBits()
189
```

The rest of the machinery associated with fingerprints can now be applied to these fragment fingerprints. For example, it's easy to find the fragments that two molecules have in common by taking the intersection of their fingerprints:

```
>>> fp2 = fpgen.GetFPForMol(ms[7],fcat)
>>> andfp = fp&fp2
>>> obl = list(andfp.GetOnBits())
>>> fcat.GetEntryDescription(obl[-1])
'ccc(cc)NC<=O>'
>>> fcat.GetEntryDescription(obl[-5])
'c<-X>ccc(N)cc'
```

or we can find the fragments that distinguish one molecule from another:

```
>>> combinedFp=fp&(fp^fp2) # can be more efficent than fp&(!fp2)
>>> obl = list(combinedFp.GetOnBits())
>>> fcat.GetEntryDescription(obl[-1])
'cccc(N)cc'
```

Or we can use the bit ranking functionality from the `rdkit.ML.InfoTheory.rdInfoTheory.InfoBitRanker` class to identify fragments that distinguish actives from inactives:

```
>>> suppl = Chem.SDMolSupplier('data/bzr.sdf')
>>> sdms = [x for x in suppl]
>>> fps = [fpgen.GetFPForMol(x,fcat) for x in sdms]
>>> from rdkit.ML.InfoTheory import InfoBitRanker
>>> ranker = InfoBitRanker(len(fps[0]),2)
>>> acts = [float(x.GetProp('ACTIVITY')) for x in sdms]
>>> for i,fp in enumerate(fps):
...   act = int(acts[i]>7)
...   ranker.AccumulateVotes(fp,act)
...
>>> top5 = ranker.GetTopN(5)
>>> for id,gain,n0,n1 in top5:
...   print int(id),'%.3f'%gain,int(n0),int(n1)
...
702 0.081 20 17
328 0.073 23 25
341 0.073 30 43
```

```
173 0.073 30 43
1034 0.069 5 53
```

The columns above are: bitId, infoGain, nInactive, nActive. Note that this approach isn't particularly effective for this artificial example.

3.12 Non-Chemical Functionality

3.12.1 Bit vectors

Bit vectors are containers for efficiently storing a set number of binary values, e.g. for fingerprints. The RDKit includes two types of fingerprints differing in how they store the values internally; the two types are easily interconverted but are best used for different purpose:

- SparseBitVects store only the list of bits set in the vector; they are well suited for storing very large, very sparsely occupied vectors like pharmacophore fingerprints. Some operations, such as retrieving the list of on bits, are quite fast. Others, such as negating the vector, are very, very slow.
- ExplicitBitVects keep track of both on and off bits. They are generally faster than SparseBitVects, but require more memory to store.

3.12.2 Discrete value vectors

3.12.3 3D grids

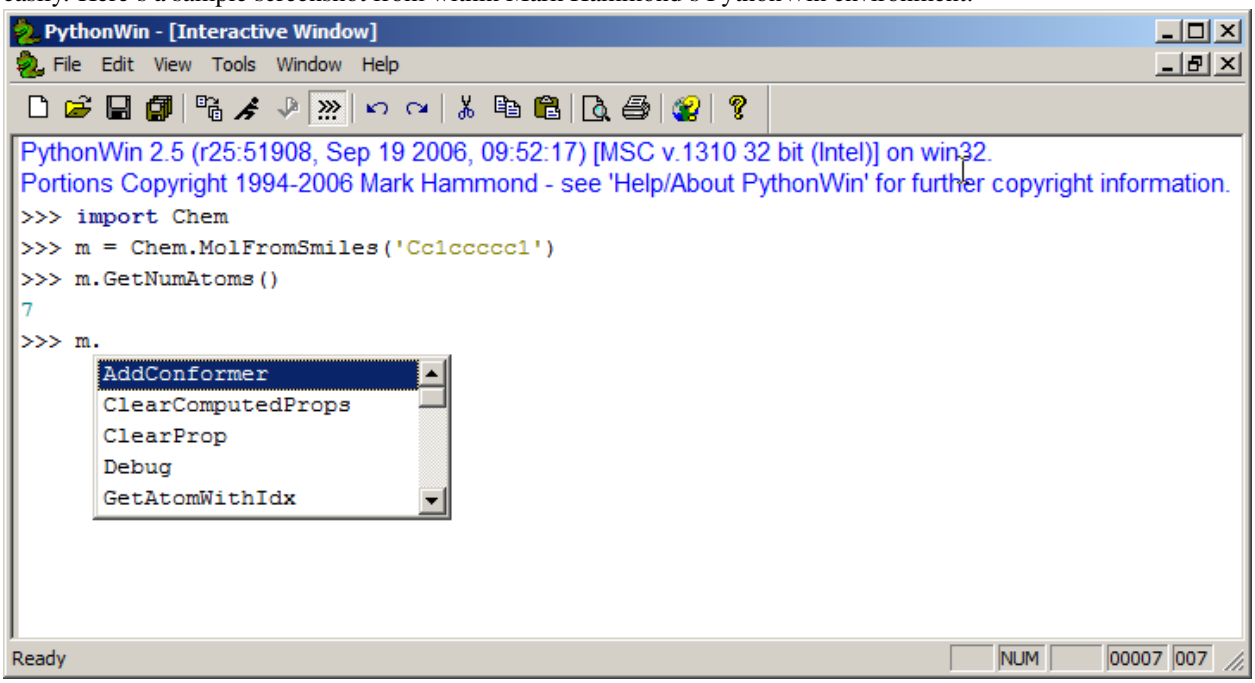
3.12.4 Points

3.13 Getting Help

There is a reasonable amount of documentation available within from the RDKit's docstrings. These are accessible using Python's help command:

```
>>> m = Chem.MolFromSmiles('Cc1ccccc1')
>>> m.GetNumAtoms()
7
>>> help(m.GetNumAtoms)
Help on method GetNumAtoms:

GetNumAtoms(...) method of rdkit.Chem.rdchem.Mol instance
    GetNumAtoms( (Mol)arg1 [, (int)onlyHeavy=-1 [, (bool)onlyExplicit=True]]) -> int :
        Returns the number of atoms in the molecule.

          ARGUMENTS:
            - onlyExplicit: (optional) include only explicit atoms (atoms in the molecular graph)
                            defaults to 1.
          NOTE: the onlyHeavy argument is deprecated


        C++ signature :
            int GetNumAtoms(RDKit::ROMol [,int=-1 [,bool=True]])

>>> m.GetNumAtoms(onlyExplicit=False)
15
```

When working in an environment that does command completion or tooltips, one can see the available methods quite easily. Here's a sample screenshot from within Mark Hammond's PythonWin environment:



3.14 Advanced Topics/Warnings

3.14.1 Editing Molecules

Some of the functionality provided allows molecules to be edited "in place":

```
>>> m = Chem.MolFromSmiles('c1ccccc1')
>>> m.GetAtomWithIdx(0).SetAtomicNum(7)
>>> Chem.SanitizeMol(m)
rdkit.Chem.rdmolops.SanitizeFlags.SANITIZE_NONE
>>> Chem.MolToSmiles(m)
'c1ccncc1'
```

Do not forget the sanitization step, without it one can end up with results that look ok (so long as you don't think):

```
>>> m = Chem.MolFromSmiles('c1ccccc1')
>>> m.GetAtomWithIdx(0).SetAtomicNum(8)
>>> Chem.MolToSmiles(m)
'c1ccocc1'
```

but that are, of course, complete nonsense, as sanitization will indicate:

```
>>> Chem.SanitizeMol(m)
Traceback (most recent call last):
  File "/usr/lib/python2.6/doctest.py", line 1253, in __run
    compileflags, 1) in test.globs
  File "<doctest default[0]>", line 1, in <module>
    Chem.SanitizeMol(m)
ValueError: Sanitization error: Can't kekulize mol
```

More complex transformations can be carried out using the `rdkit.Chem.rdchem.EditableMol` class:

```
>>> m = Chem.MolFromSmiles('CC(=O)O')
>>> em = Chem.EditableMol(m)
>>> em.ReplaceAtom(3,Chem.Atom(7))
>>> em.AddAtom(Chem.Atom(6))
4
>>> em.AddAtom(Chem.Atom(6))
5
>>> em.AddBond(3,4,Chem.BondType.SINGLE)
4
>>> em.AddBond(4,5,Chem.BondType.DOUBLE)
5
>>> em.RemoveAtom(0)
```

Note that the `rdkit.Chem.rdchem.EditableMol` must be converted back into a standard `rdkit.Chem.rdchem.Mol` before much else can be done with it:

```
>>> em.GetNumAtoms()
Traceback (most recent call last):
  File "/usr/lib/python2.6/doctest.py", line 1253, in __run
    compileflags, 1) in test.globs
  File "<doctest default[0]>", line 1, in <module>
    em.GetNumAtoms()
AttributeError: 'EditableMol' object has no attribute 'GetNumAtoms'
>>> Chem.MolToSmiles(em)
Traceback (most recent call last):
  File "/usr/lib/python2.6/doctest.py", line 1253, in __run
    compileflags, 1) in test.globs
  File "<doctest default[1]>", line 1, in <module>
    Chem.MolToSmiles(em)
ArgumentError: Python argument types in
    rdkit.Chem.rdmolfiles.MolToSmiles(EditableMol)
did not match C++ signature:
    MolToSmiles(RDKit::ROMol {lvalue} mol, bool isomericSmiles=False, bool kekuleSmiles=False, int ro
>>> m2 = em.GetMol()
>>> Chem.SanitizeMol(m2)
rdkit.Chem.rdmolops.SanitizeFlags.SANITIZE_NONE
>>> Chem.MolToSmiles(m2)
'C=CNC=O'
```

It is even easier to generate nonsense using the EditableMol than it is with standard molecules. If you need chemically reasonable results, be certain to sanitize the results.

3.15 Miscellaneous Tips and Hints

3.15.1 Chem vs AllChem

The majority of "basic" chemical functionality (e.g. reading/writing molecules, substructure searching, molecular cleanup, etc.) is in the `rdkit.Chem` module. More advanced, or less frequently used, functionality is in `rdkit.Chem.AllChem`. The distinction has been made to speed startup and lower import times; there's no sense in loading the 2D->3D library and force field implementation if one is only interested in reading and writing a couple of molecules. If you find the Chem/AllChem thing annoying or confusing, you can use python's "import ... as ..." syntax to remove the irritation:

```
>>> from rdkit.Chem import AllChem as Chem
>>> m = Chem.MolFromSmiles('CCC')
```

3.15.2 The SSSR Problem

As others have ranted about with more energy and eloquence than I intend to, the definition of a molecule's smallest set of smallest rings is not unique. In some high symmetry molecules, a "true" SSSR will give results that are unappealing. For example, the SSSR for cubane only contains 5 rings, even though there are "obviously" 6. This problem can be fixed by implementing a *small* (instead of *smallest*) set of smallest rings algorithm that returns symmetric results. This is the approach that we took with the RDKit.

Because it is sometimes useful to be able to count how many SSSR rings are present in the molecule, there is a `rdkit.Chem.rdmolops.GetSSSR` function, but this only returns the SSSR count, not the potentially non-unique set of rings.

3.16 List of Available Descriptors

Descriptor/Descriptor Family	Notes
Gasteiger/Marsili Partial Charges	*Tetrahedron* **36**:3219-28 (1980)
BalabanJ	*Chem. Phys. Lett.* **89**:399-404 (1982)
BertzCT	*J. Am. Chem. Soc.* **103**:3599-601 (1981)
Ipc	*J. Chem. Phys.* **67**:4517-33 (1977)
HallKierAlpha	*Rev. Comput. Chem.* **2**:367-422 (1991)
Kappa1 - Kappa3	*Rev. Comput. Chem.* **2**:367-422 (1991)
Chi0, Chi1	*Rev. Comput. Chem.* **2**:367-422 (1991)
Chi0n - Chi4n	*Rev. Comput. Chem.* **2**:367-422 (1991)
Chi0v - Chi4v	*Rev. Comput. Chem.* **2**:367-422 (1991)
MolLogP	Wildman and Crippen *JCICS* **39**:868-73 (1999)
MolMR	Wildman and Crippen *JCICS* **39**:868-73 (1999)
MolWt	
HeavyAtomCount	
HeavyAtomMolWt	
NHOHCount	
NOCount	
NumHAcceptors	
NumHDonors	
NumHeteroatoms	
NumRotatableBonds	
NumValenceElectrons	
RingCount	
TPSA	*J. Med. Chem.* **43**:3714-7, (2000)
LabuteASA	*J. Mol. Graph. Mod.* **18**:464-77 (2000)
PEOE_VSA1 - PEOE_VSA14	MOE-type descriptors using partial charges and surface area contributionshttp://www.chemcomp.
SMR_VSA1 - SMR_VSA10	MOE-type descriptors using MR contributions and surface area contributionshttp://www.chemcor
SlogP_VSA1 - SlogP_VSA12	MOE-type descriptors using LogP contributions and surface area contributionshttp://www.chemc
EState_VSA1 - EState_VSA11	MOE-type descriptors using EState indices and surface area contributions (developed at RD, not c
VSA_EState1 - VSA_EState10	MOE-type descriptors using EState indices and surface area contributions (developed at RD, not c
Topliss fragments	implemented using a set of SMARTS definitions in $(RDBASE)/Data/FragmentDescriptors.csv

3.17 List of Available Fingerprints

Fingerprint Type	Notes
Topological	a Daylight-like fingerprint based on hashing molecular subgraphs
Atom Pairs	*JCICS* **25**:64-73 (1985)
Topological Torsions	*JCICS* **27**:82-5 (1987)
MACCS keys	Using the 166 public keys implemented as SMARTS
Morgan/Circular	Fingerprints based on the Morgan algorithm, similar to the ECFP fingerprint*JCIM* **50**:742-54 (2010).
2D Pharmacophore	Uses topological distances between pharmacophoric points.

3.18 Feature Definitions Used in the Morgan Fingerprints

These are adapted from the definitions in Gobbi, A. & Poppinger, D. "Genetic optimization of combinatorial libraries." *Biotechnology and Bioengineering* **61**, 47-54 (1998).

Feature	SMARTS
Donor	`[$([N;!H0;v3,v4&+1]),$([O,S;H1;+0]),n&H1&+0]`
Acceptor	`[$([O,S;H1;v2;!$(*-*=[O,N,P,S])]),$([O,S;H0;v2]),$([O,S;-]),$([N;v3;!$(N-*=[O,N,P,S`
Aromatic	`[a]`
Halogen	`[F,Cl,Br,I]`
Basic	`[#7;+,$([N;H2&+0][$([C,a]);!$([C,a](=O))]),$([N;H1&+0]([$([C,a]);!$([C,a](=O))])[$(`
Acidic	`[$([C,S](=[O,S,P])-[O;H1,-1])]`

3.19 License

CHAPTER

FOUR

THE RDKIT BOOK

4.1 Misc Cheminformatics Topics

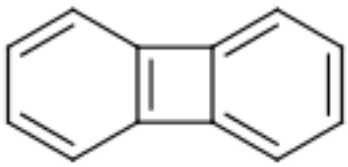
4.1.1 Aromaticity

Aromaticity is one of those unpleasant topics that is simultaneously simple and impossibly complicated. Since neither experimental nor theoretical chemists can agree with each other about a definition, it's necessary to pick something arbitrary and stick to it. This is the approach taken in the RDKit.

Instead of using patterns to match known aromatic systems, the aromaticity perception code in the RDKit uses a set of rules. The rules are relatively straightforward.

Aromaticity is a property of atoms and bonds in rings. An aromatic bond must be between aromatic atoms, but a bond between aromatic atoms does not need to be aromatic.

For example the fusing bonds here are not considered to be aromatic by the RDKit:



```
>>> from rdkit import Chem
>>> m = Chem.MolFromSmiles('C1=CC2=C(C=C1)C1=CC=CC=C21')
>>> m.GetAtomWithIdx(3).GetIsAromatic()
True
>>> m.GetAtomWithIdx(6).GetIsAromatic()
True
>>> m.GetBondBetweenAtoms(3,6).GetIsAromatic()
False
```

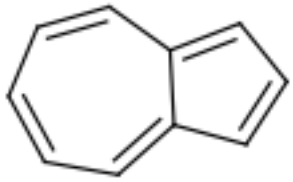
A ring, or fused ring system, is considered to be aromatic if it obeys the 4N+2 rule. Contributions to the electron count are determined by atom type and environment. Some examples:

Fragment	Number of pi electrons
c(a)a	1
n(a)a	1
An(a)a	2
o(a)a	2
s(a)a	2
se(a)a	2
te(a)a	2
O=c(a)a	0
N=c(a)a	0
*(a)a	0, 1, or 2

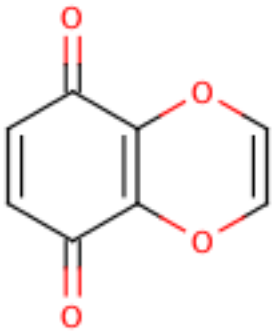
Notation a: any aromatic atom; A: any atom, include H; *: a dummy atom

Notice that exocyclic bonds to electronegative atoms "steal" the valence electron from the ring atom and that dummy atoms contribute whatever count is necessary to make the ring aromatic.

The use of fused rings for aromaticity can lead to situations where individual rings are not aromatic, but the fused system is. An example of this is azulene:



An extreme example, demonstrating both fused rings and the influence of exocyclic double bonds:




```
>>> m=Chem.MolFromSmiles('O=C1C=CC(=O)C2=C1OC=CO2')
>>> m.GetAtomWithIdx(6).GetIsAromatic()
True
>>> m.GetAtomWithIdx(7).GetIsAromatic()
True
>>> m.GetBondBetweenAtoms(6,7).GetIsAromatic()
False
```

Note: For reasons of computation expediency, aromaticity perception is only done for fused-ring systems where all members are at most 24 atoms in size.

4.1.2 Ring Finding and SSSR

[Section taken from "Getting Started" document]

As others have ranted about with more energy and eloquence than I intend to, the definition of a molecule's smallest set of smallest rings is not unique. In some high symmetry molecules, a "true" SSSR will give results that are unappealing. For example, the SSSR for cubane only contains 5 rings, even though there are "obviously" 6. This problem can be fixed by implementing a *small* (instead of *smallest*) set of smallest rings algorithm that returns symmetric results. This is the approach that we took with the RDKit.

Because it is sometimes useful to be able to count how many SSSR rings are present in the molecule, there is a GetSSSR function, but this only returns the SSSR count, not the potentially non-unique set of rings.

4.2 Chemical Reaction Handling

4.2.1 Reaction SMARTS

Not SMIRKS [1] , not reaction SMILES [2], derived from SMARTS [3].

The general grammar for a reaction SMARTS is :

```
reaction   ::=   reactants ">>" products
reactants  ::=   molecules
products   ::=   molecules
molecules  ::=   molecule
                 molecules "."  molecule
molecule   ::=   a valid SMARTS string without "."  characters
```

Some features

Mapped dummy atoms in the product template are replaced by the corresponding atom in the reactant:

```
>>> from rdkit.Chem import AllChem
>>> rxn = AllChem.ReactionFromSmarts('[C:1]=[O,N:2]>>[C:1][*:2]')
>>> [Chem.MolToSmiles(x,1) for x in rxn.RunReactants((Chem.MolFromSmiles('CC=O'),))[0]]
['CCO']
>>> [Chem.MolToSmiles(x,1) for x in rxn.RunReactants((Chem.MolFromSmiles('CC=N'),))[0]]
['CCN']
```

but unmapped dummy atoms are left as dummies:

```
>>> rxn = AllChem.ReactionFromSmarts('[C:1]=[O,N:2]>>[*][C:1][*:2]')
>>> [Chem.MolToSmiles(x,1) for x in rxn.RunReactants((Chem.MolFromSmiles('CC=O'),))[0]]
['[*]C(C)O']
```

"Any" bonds in the products are replaced by the corresponding bond in the reactant:

```
>>> rxn = AllChem.ReactionFromSmarts('[C:1]~[O,N:2]>>[*][C:1]~[*:2]')
>>> [Chem.MolToSmiles(x,1) for x in rxn.RunReactants((Chem.MolFromSmiles('C=O'),))[0]]
['[*]C=O']
>>> [Chem.MolToSmiles(x,1) for x in rxn.RunReactants((Chem.MolFromSmiles('CO'),))[0]]
```

[1] http://www.daylight.com/dayhtml/doc/theory/theory.smirks.html
[2] http://www.daylight.com/dayhtml/doc/theory/theory.smiles.html
[3] http://www.daylight.com/dayhtml/doc/theory/theory.smarts.html

```
['[*]CO']
>>> [Chem.MolToSmiles(x,1) for x in rxn.RunReactants((Chem.MolFromSmiles('C#N'),))[0]]
['[*]C#N']
```

Rules and caveats

1. Include atom map information at the end of an atom query. So do [C,N,O:1] or [C;R:1].
2. Don't forget that unspecified bonds in SMARTS are either single or aromatic. Bond orders in product templates are assigned when the product template itself is constructed and it's not always possible to tell if the bond should be single or aromatic:

```
>>> rxn = AllChem.ReactionFromSmarts('[#6:1][#7,#8:2]>>[#6:1][#6:2]')
>>> [Chem.MolToSmiles(x,1) for x in rxn.RunReactants((Chem.MolFromSmiles('C1NCCCC1'),))[0]]
['C1CCCCC1']
>>> [Chem.MolToSmiles(x,1) for x in rxn.RunReactants((Chem.MolFromSmiles('c1ncccc1'),))[0]]
['c1cccc-c1']
```

So if you want to copy the bond order from the reactant, use an "Any" bond:

```
>>> rxn = AllChem.ReactionFromSmarts('[#6:1][#7,#8:2]>>[#6:1]~[#6:2]')
>>> [Chem.MolToSmiles(x,1) for x in rxn.RunReactants((Chem.MolFromSmiles('c1ncccc1'),))[0]]
['c1ccccc1']
```

4.3 The Feature Definition File Format

An FDef file contains all the information needed to define a set of chemical features. It contains definitions of feature types that are defined from queries built up using Daylight's SMARTS language. [3] The FDef file can optionally also include definitions of atom types that are used to make feature definitions more readable.

4.3.1 Chemical Features

Chemical features are defined by a Feature Type and a Feature Family. The Feature Family is a general classification of the feature (such as "Hydrogen-bond Donor" or "Aromatic") while the Feature Type provides additional, higher-resolution, information about features. Pharmacophore matching is done using Feature Family's. Each feature type contains the following pieces of information:

- A SMARTS pattern that describes atoms (one or more) matching the feature type.
- Weights used to determine the feature's position based on the positions of its defining atoms.

4.3.2 Syntax of the FDef file

AtomType definitions

An AtomType definition allows you to assign a shorthand name to be used in place of a SMARTS string defining an atom query. This allows FDef files to be made much more readable. For example, defining a non-polar carbon atom like this:

```
AtomType Carbon_NonPolar [C&!$(C=[O,N,P,S])&!$(C#N)]
```

creates a new name that can be used anywhere else in the FDef file that it would be useful to use this SMARTS. To reference an AtomType, just include its name in curly brackets. For example, this excerpt from an FDef file defines another atom type - Hphobe - which references the Carbon_NonPolar definition:

```
AtomType Carbon_NonPolar [C&!$(C=[O,N,P,S])&!$(C#N)]
AtomType Hphobe [{Carbon_NonPolar},c,s,S&H0&v2,F,Cl,Br,I]
```

Note that `{Carbon_NonPolar}` is used in the new AtomType definition without any additional decoration (no square brackes or recursive SMARTS markers are required).

Repeating an AtomType results in the two definitions being combined using the SMARTS ”,” (or) operator. Here's an example:

```
AtomType d1 [N&!H0]
AtomType d1 [O&!H0]
```

This is equivalent to:

```
AtomType d1 [N&!H0,O&!H0]
```

Which is equivalent to the more efficient:

```
AtomType d1 [N,O;!H0]
```

Note that these examples tend to use SMARTS's high-precendence and operator “&” and not the low-precedence and ”;”. This can be important when AtomTypes are combined or when they are repeated. The SMARTS ”,” operator is higher precedence than ”;”, so definitions that use ”;” can lead to unexpected results.

It is also possible to define negative AtomType queries:

```
AtomType d1 [N,O,S]
AtomType !d1 [H0]
```

The negative query gets combined with the first to produce a definition identical to this:

```
AtomType d1 [!H0;N,O,S]
```

Note that the negative AtomType is added to the beginning of the query.

Feature definitions

A feature definition is more complex than an AtomType definition and stretches across multiple lines:

```
DefineFeature HDonor1 [N,O;!H0]
Family HBondDonor
Weights 1.0
EndFeature
```

The first line of the feature definition includes the feature type and the SMARTS string defining the feature. The next two lines (order not important) define the feature's family and its atom weights (a comma-delimited list that is the same length as the number of atoms defining the feature). The atom weights are used to calculate the feature's locations based on a weighted average of the positions of the atom defining the feature. More detail on this is provided below. The final line of a feature definition must be EndFeature. It is perfectly legal to mix AtomType definitions with feature definitions in the FDef file. The one rule is that AtomTypes must be defined before they are referenced.

Additional syntax notes:

- Any line that begins with a # symbol is considered a comment and will be ignored.
- A backslash character, , at the end of a line is a continuation character, it indicates that the data from that line is continued on the next line of the file. Blank space at the beginning of these additional lines is ignored. For example, this AtomType definition:

```
AtomType tButylAtom [$([C;!R](-[CH3])(-[CH3])(-[CH3])),\
$([CH3](-[C;!R](-[CH3])(-[CH3])))]
```

is exactly equivalent to this one:

```
AtomType tButylAtom [$([C;!R](-[CH3])(-[CH3])(-[CH3])),$([CH3](-[C;!R](-[CH3])(-[CH3])))]
```

(though the first form is much easier to read!)

Atom weights and feature locations

4.3.3 Frequently Asked Question(s)

- What happens if a Feature Type is repeated in the file? Here's an example:

```
DefineFeature HDonor1 [O&!H0]
Family HBondDonor
Weights 1.0
EndFeature

DefineFeature HDonor1 [N&!H0]
Family HBondDonor
Weights 1.0
EndFeature
```

In this case both definitions of the HDonor1 feature type will be active. This is functionally identical to:

```
DefineFeature HDonor1 [O,N;!H0]
Family HBondDonor
Weights 1.0
EndFeature
```

However the formulation of this feature definition with a duplicated feature type is considerably less efficient and more confusing than the simpler combined definition.

4.4 Representation of Pharmacophore Fingerprints

In the RDKit scheme the bit ids in pharmacophore fingerprints are not hashed: each bit corresponds to a particular combination of features and distances. A given bit id can be converted back to the corresponding feature types and distances to allow interpretation. An illustration for 2D pharmacophores is shown in *Figure 1: Bit numbering in pharmacophore fingerprints*.

4.5 Atom-Atom Matching in Substructure Queries

When doing substructure matches for queries derived from SMARTS the rules for which atoms in the molecule should match which atoms in the query are well defined.[#smarts]_ The same is not necessarily the case when the query molecule is derived from a mol block or SMILES.

The general rule used in the RDKit is that if you don't specify a property in the query, then it's not used as part of the matching criteria and that Hs are ignored. This leads to the following behavior:

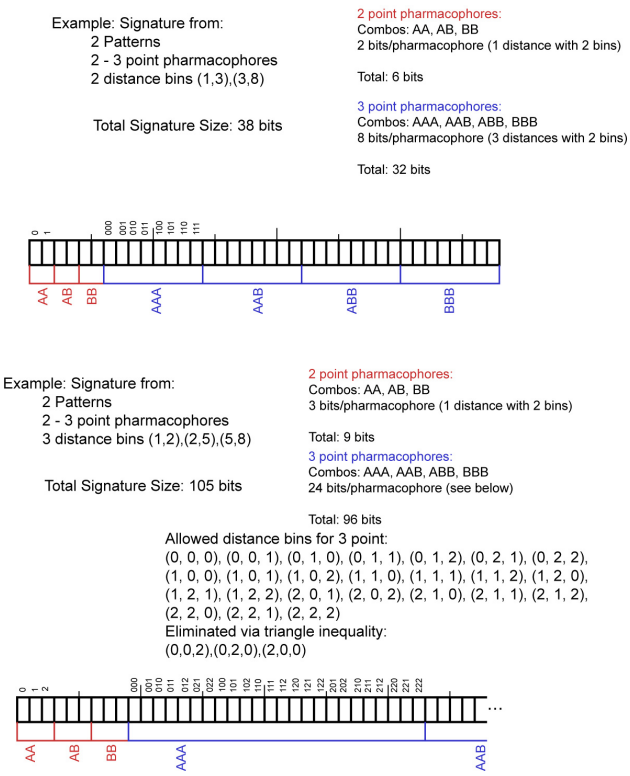


Figure 4.1: Figure 1: Bit numbering in pharmacophore fingerprints

Molecule	Query	Match
CCO	CCO	Yes
CC[O-]	CCO	Yes
CCO	CC[O-]	No
CC[O-]	CC[O-]	Yes
CC[O-]	CC[OH]	Yes
CCOC	CC[OH]	Yes
CCOC	CCO	Yes
CCC	CCC	Yes
CC[14C]	CCC	Yes
CCC	CC[14C]	No
CC[14C]	CC[14C]	Yes
OCO	C	Yes
OCO	[CH]	Yes
OCO	[CH2]	Yes
OCO	[CH3]	Yes
O[CH2]O	C	Yes
O[CH2]O	[CH2]	Yes

4.6 License

CHAPTER

FIVE

RDKIT COOKBOOK

5.1 What is this?

This document provides examples of how to carry out particular tasks using the RDKit functionality from Python. The contents have been contributed by the RDKit community.

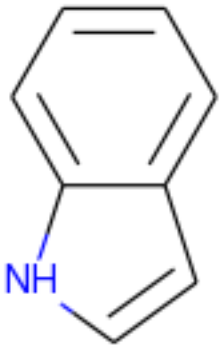
If you find mistakes, or have suggestions for improvements, please either fix them yourselves in the source document (the .rst file) or send them to the mailing list: rdkit-discuss@lists.sourceforge.net (you will need to subscribe first)

5.2 Miscellaneous Topics

5.2.1 Using a different aromaticity model

By default, the RDKit applies its own model of aromaticity (explained in the RDKit Theory Book) when it reads in molecules. It is, however, fairly easy to override this and use your own aromaticity model.

The easiest way to do this is it provide the molecules as SMILES with the aromaticity set as you would prefer to have it. For example, consider indole:



By default the RDKit considers both rings to be aromatic:

```
>>> from rdkit import Chem
>>> m = Chem.MolFromSmiles('N1C=Cc2ccccc12')
>>> m.GetSubstructMatches(Chem.MolFromSmarts('c'))
((1,), (2,), (3,), (4,), (5,), (6,), (7,), (8,))
```

If you'd prefer to treat the five-membered ring as aliphatic, which is how the input SMILES is written, you just need to do a partial sanitization that skips the kekulization and aromaticity perception steps:

```
>>> m2 = Chem.MolFromSmiles('N1C=Cc2ccccc12',sanitize=False)
>>> Chem.SanitizeMol(m2,sanitizeOps=Chem.SanitizeFlags.SANITIZE_ALL^Chem.SanitizeFlags.SANITIZE_KEKU
rdkit.Chem.rdmolops.SanitizeFlags.SANITIZE_NONE
>>> m2.GetSubstructMatches(Chem.MolFromSmarts('c'))
((3,), (4,), (5,), (6,), (7,), (8,))
```

It is, of course, also possible to write your own aromaticity perception function, but that is beyond the scope of this document.

5.3 Manipulating Molecules

5.3.1 Cleaning up heterocycles

Mailing list discussions:

- http://www.mail-archive.com/rdkit-discuss@lists.sourceforge.net/msg01185.html
- http://www.mail-archive.com/rdkit-discuss@lists.sourceforge.net/msg01162.html
- http://www.mail-archive.com/rdkit-discuss@lists.sourceforge.net/msg01900.html

- http://www.mail-archive.com/rdkit-discuss@lists.sourceforge.net/msg01901.html

The code:

```
""" sanifix4.py

  Contribution from James Davidson
"""
from rdkit import Chem
from rdkit.Chem import AllChem

def _FragIndicesToMol(oMol,indices):
    em = Chem.EditableMol(Chem.Mol())

    newIndices={}
    for i,idx in enumerate(indices):
        em.AddAtom(oMol.GetAtomWithIdx(idx))
        newIndices[idx]=i

    for i,idx in enumerate(indices):
        at = oMol.GetAtomWithIdx(idx)
        for bond in at.GetBonds():
            if bond.GetBeginAtomIdx()==idx:
                oidx = bond.GetEndAtomIdx()
            else:
                oidx = bond.GetBeginAtomIdx()
            # make sure every bond only gets added once:
            if oidx<idx:
                continue
            em.AddBond(newIndices[idx],newIndices[oidx],bond.GetBondType())
    res = em.GetMol()
    res.ClearComputedProps()
    Chem.GetSymmSSSR(res)
    res.UpdatePropertyCache(False)
    res._idxMap=newIndices
    return res

def _recursivelyModifyNs(mol,matches,indices=None):
    if indices is None:
        indices=[]
    res=None
    while len(matches) and res is None:
        tIndices=indices[:]
        nextIdx = matches.pop(0)
        tIndices.append(nextIdx)
        nm = Chem.Mol(mol.ToBinary())
        nm.GetAtomWithIdx(nextIdx).SetNoImplicit(True)
        nm.GetAtomWithIdx(nextIdx).SetNumExplicitHs(1)
        cp = Chem.Mol(nm.ToBinary())
        try:
            Chem.SanitizeMol(cp)
        except ValueError:
            res,indices = _recursivelyModifyNs(nm,matches,indices=tIndices)
        else:
            indices=tIndices
            res=cp
    return res,indices

def AdjustAromaticNs(m,nitrogenPattern='[n&D2&H0;r5,r6]'):
```

```
    """
       default nitrogen pattern matches Ns in 5 rings and 6 rings in order to be able
       to fix: O=c1ccncc1
    """
    Chem.GetSymmSSSR(m)
    m.UpdatePropertyCache(False)

    # break non-ring bonds linking rings:
    em = Chem.EditableMol(m)
    linkers = m.GetSubstructMatches(Chem.MolFromSmarts('[r]!@[r]'))
    plsFix=set()
    for a,b in linkers:
        em.RemoveBond(a,b)
        plsFix.add(a)
        plsFix.add(b)
    nm = em.GetMol()
    for at in plsFix:
        at=nm.GetAtomWithIdx(at)
        if at.GetIsAromatic() and at.GetAtomicNum()==7:
            at.SetNumExplicitHs(1)
            at.SetNoImplicit(True)

    # build molecules from the fragments:
    fragLists = Chem.GetMolFrags(nm)
    frags = [_FragIndicesToMol(nm,x) for x in fragLists]

    # loop through the fragments in turn and try to aromatize them:
    ok=True
    for i,frag in enumerate(frags):
        cp = Chem.Mol(frag.ToBinary())
        try:
            Chem.SanitizeMol(cp)
        except ValueError:
            matches = [x[0] for x in frag.GetSubstructMatches(Chem.MolFromSmarts(nitrogenPattern))]
            lres,indices=_recursivelyModifyNs(frag,matches)
            if not lres:
                #print 'frag %d failed (%s)'%(i,str(fragLists[i]))
                ok=False
                break
            else:
                revMap={}
                for k,v in frag._idxMap.iteritems():
                    revMap[v]=k
                for idx in indices:
                    oatom = m.GetAtomWithIdx(revMap[idx])
                    oatom.SetNoImplicit(True)
                    oatom.SetNumExplicitHs(1)
    if not ok:
        return None
    return m
```

Examples of using it:

```
smis= ('O=c1ccc2ccccc2n1',
       'Cc1nnnn1C',
       'CCc1ccc2nc(=O)c(cc2c1)Cc1nnnn1C1CCCCC1',
       'c1cnc2cc3ccnc3cc12',
       'c1cc2cc3ccnc3cc2n1',
       'O=c1ccnc(c1)-c1cnc2cc3ccnc3cc12',
```

```
       'O=c1ccnc(c1)-c1cc1',
       )
for smi in smis:
      m = Chem.MolFromSmiles(smi,False)
      try:
          m.UpdatePropertyCache(False)
          cp = Chem.Mol(m.ToBinary())
          Chem.SanitizeMol(cp)
          m = cp
          print 'fine:',Chem.MolToSmiles(m)
      except ValueError:
          nm=AdjustAromaticNs(m)
          if nm is not None:
              Chem.SanitizeMol(nm)
              print 'fixed:',Chem.MolToSmiles(nm)
          else:
              print 'still broken:',smi
```

This produces:

```
fixed: O=c1ccc2ccccc2[nH]1
fine: Cc1nnnn1C
fixed: CCc1ccc2[nH]c(=O)c(Cc3nnnn3C3CCCCC3)cc2c1
fine: C1=Cc2cc3c(cc2=N1)C=CN=3
fine: C1=Cc2cc3c(cc2=N1)N=CC=3
fixed: O=c1cc[nH]c(C2=CN=c3cc4c(cc32)=NC=C4)c1
still broken: O=c1ccnc(c1)-c1cc1
```

5.3.2 Parallel conformation generation

Mailing list discussion: http://www.mail-archive.com/rdkit-discuss@lists.sourceforge.net/msg02648.html

The code:

```
""" contribution from Andrew Dalke """
import sys
from rdkit import Chem
from rdkit.Chem import AllChem

# Download this from http://pypi.python.org/pypi/futures
from concurrent import futures

# Download this from http://pypi.python.org/pypi/progressbar
import progressbar

## On my machine, it takes 39 seconds with 1 worker and 10 seconds with 4.
## 29.055u 0.102s 0:28.68 101.6%   0+0k 0+3io 0pf+0w
#max_workers=1

## With 4 threads it takes 11 seconds.
## 34.933u 0.188s 0:10.89 322.4%   0+0k 125+1io 0pf+0w
max_workers=4

# (The "u"ser time includes time spend in the children processes.
#  The wall-clock time is 28.68 and 10.89 seconds, respectively.)

# This function is called in the subprocess.
# The parameters (molecule and number of conformers) are passed via a Python
```

```
def generateconformations(m, n):
    m = Chem.AddHs(m)
    ids=AllChem.EmbedMultipleConfs(m, numConfs=n)
    for id in ids:
        AllChem.UFFOptimizeMolecule(m, confId=id)
    # EmbedMultipleConfs returns a Boost-wrapped type which
    # cannot be pickled. Convert it to a Python list, which can.
    return m, list(ids)

smi_input_file, sdf_output_file = sys.argv[1:3]

n = int(sys.argv[3])

writer = Chem.SDWriter(sdf_output_file)

suppl = Chem.SmilesMolSupplier(smi_input_file, titleLine=False)

with futures.ProcessPoolExecutor(max_workers=max_workers) as executor:
    # Submit a set of asynchronous jobs
    jobs = []
    for mol in suppl:
        if mol:
            job = executor.submit(generateconformations, mol, n)
            jobs.append(job)

    widgets = ["Generating conformations; ", progressbar.Percentage(), " ",
               progressbar.ETA(), " ", progressbar.Bar()]
    pbar = progressbar.ProgressBar(widgets=widgets, maxval=len(jobs))
    for job in pbar(futures.as_completed(jobs)):
        mol,ids=job.result()
        for id in ids:
            writer.write(mol, confId=id)
writer.close()
```

5.3.3 Neutralizing Charged Molecules

Mailing list discussion: http://www.mail-archive.com/rdkit-discuss@lists.sourceforge.net/msg02648.html

Wiki page: http://code.google.com/p/rdkit/wiki/NeutralisingCompounds

The code:

```
""" contribution from Hans de Winter """
from rdkit import Chem
from rdkit.Chem import AllChem

def _InitialiseNeutralisationReactions():
    patts= (
        # Imidazoles
        ('[n+;H]','n'),
        # Amines
        ('[N+;!H0]','N'),
        # Carboxylic acids and alcohols
        ('[$([O-]);!$([O-][#7])]','O'),
        # Thiols
        ('[S-;X1]','S'),
        # Sulfonamides
        ('[$([N-;X2]S(=O)=O)]','N'),
```

```
        # Enamines
        ('[$([N-;X2][C,N]=C)]','N'),
        # Tetrazoles
        ('[n-]','[nH]'),
        # Sulfoxides
        ('[$([S-]=O)]','S'),
        # Amides
        ('[$([N-]C=O)]','N'),
        )
    return [(Chem.MolFromSmarts(x),Chem.MolFromSmiles(y,False)) for x,y in patts]

_reactions=None
def NeutraliseCharges(smiles, reactions=None):
    global _reactions
    if reactions is None:
        if _reactions is None:
            _reactions=_InitialiseNeutralisationReactions()
        reactions=_reactions
    mol = Chem.MolFromSmiles(smiles)
    replaced = False
    for i,(reactant, product) in enumerate(reactions):
        while mol.HasSubstructMatch(reactant):
            replaced = True
            rms = AllChem.ReplaceSubstructs(mol, reactant, product)
            mol = rms[0]
    if replaced:
        return (Chem.MolToSmiles(mol,True), True)
    else:
        return (smiles, False)
```

Examples of using it:

```
smis=("c1cccc[nH+]1",
      "C[N+](C)(C)C","c1ccccc1[NH3+]",
      "CC(=O)[O-]","c1ccccc1[O-]",
      "CCS",
      "C[N-]S(=O)(=O)C",
      "C[N-]C=C","C[N-]N=C",
      "c1ccc[n-]1",
      "CC[N-]C(=O)CC")
for smi in smis:
    (molSmiles, neutralised) = NeutraliseCharges(smi)
    print smi,"->",molSmiles
```

This produces:

```
c1cccc[nH+]1 -> c1ccncc1
C[N+](C)(C)C -> C[N+](C)(C)C
c1ccccc1[NH3+] -> Nc1ccccc1
CC(=O)[O-] -> CC(=O)O
c1ccccc1[O-] -> Oc1ccccc1
CCS -> CCS
C[N-]S(=O)(=O)C -> CNS(C)(=O)=O
C[N-]C=C -> C=CNC
C[N-]N=C -> C=NNC
c1ccc[n-]1 -> c1cc[nH]c1
CC[N-]C(=O)CC -> CCNC(=O)CC
```

5.4 License

CHAPTER

SIX

THE RDKIT DATABASE CARTRIDGE

6.1 What is this?

This document is a tutorial and reference guide for the RDKit PostgreSQL cartridge.

If you find mistakes, or have suggestions for improvements, please either fix them yourselves in the source document (the .rst file) or send them to the mailing list: rdkit-discuss@lists.sourceforge.net (you will need to subscribe first)

6.2 Tutorial

6.2.1 Introduction

In this example I show how to load a database from the SMILES file of commercially available compounds that is downloadable from emolecules.com at URL http://www.emolecules.com/doc/plus/download-database.php

If you choose to repeat this exact example yourself, please note that it takes several hours to load the 6 million row database and generate all fingerprints. The timing information below was collected on a commodity desktop PC (Dell Studio XPS with a 2.9GHz i7 CPU and 8GB of RAM) running Ubuntu 12.04 and using PostgreSQL v9.1.4. The database was installed with default parameters.

To improve performance while loading the database and building the index, I changed a couple of postgres configuration settings in *postgresql.conf*

```
fsync = off                             # turns forced synchronization on or off
synchronous_commit = off                # immediate fsync at commit
full_page_writes = off                          # recover from partial page writes
```

And to improve search performance, I allowed postgresql to use more memory than the extremely conservative default settings:

```
shared_buffers = 2048MB                         # min 128kB
                                        # (change requires restart)
work_mem = 128MB                                # min 64kB
```

6.2.2 Creating the database

First create the database and install the cartridge:

```
~/RDKit_trunk/Data/emolecules > createdb emolecules
~/RDKit_trunk/Data/emolecules > psql -c 'create extension rdkit' emolecules
```

Now create and populate a table holding the raw data:

```
~/RDKit_trunk/Data/emolecules > psql -c 'create table raw_data (id SERIAL, smiles text, emol_id integ
NOTICE:  CREATE TABLE will create implicit sequence "raw_data_id_seq" for serial column "raw_data.id"
CREATE TABLE
~/RDKit_trunk/Data/emolecules > zcat emolecules-2013-02-01.smi.gz | sed '1d; s/\\/\\\\/g' | psql -c '
```

Create the molecule table, but only for SMILES that the RDKit accepts:

```
~/RDKit_trunk/Data/emolecules > psql emolecules
psql (9.1.4)
Type "help" for help.
emolecules=# select * into mols from (select id,mol_from_smiles(smiles::cstring) m from raw_data) tmp
WARNING:  could not create molecule from SMILES 'CN(C)C(=[N+](C)C)Cl.F[P-](F)(F)(F)(F)F'
... a lot of warnings deleted ...
SELECT 6008732
emolecules=# create index molidx on mols using gist(m);
CREATE INDEX
```

The last step is only required if you plan to do substructure searches.

6.2.3 Substructure searches

Example query molecules taken from the eMolecules home page:

```
emolecules=# select count(*) from mols where m@>'c1cccc2c1nncc2' ;
 count
-------
   1593
(1 row)

Time: 3413.018 ms
emolecules=# select count(*) from mols where m@>'c1ccnc2c1nccn2' ;
 count
-------
  3692
(1 row)

Time: 760.860 ms
emolecules=# select count(*) from mols where m@>'c1cncc2n1ccn2' ;
 count
-------
  2359
(1 row)

Time: 790.864 ms
emolecules=# select count(*) from mols where m@>'Nc1ncnc(N)n1' ;
 count
-------
 14086
(1 row)

Time: 2445.430 ms
```

Notice that the last query is starting to take a while to execute and count all the results. This is even more extreme with the next few queries:

```
emolecules=# select count(*) from mols where m@>'c1scnn1' ;
 count
-------
 108477
(1 row)

Time: 37925.126 ms
emolecules=# select count(*) from mols where m@>'c1cccc2c1CNCCN2' ;
 count
-------
  2490
(1 row)

Time: 46126.816 ms
emolecules=# select count(*) from mols where m@>'c1cccc2c1ncs2' ;
 count
-------
 104895
(1 row)

Time: 77505.272 ms
```

Given we're searching through 6 million compounds these search times aren't incredibly slow, but it would be nice to have them quicker.

One easy way to speed things up, particularly for queries that return a large number of results, is to only retrieve a limited number of results:

```
emolecules=# select * from mols where m@>'c1cccc2c1ncs2' limit 100 ;
   id    |                                m
---------+-------------------------------------------------------------
 5273717 | OC1CC(Nc2nc3ccccc3s2)C1
 5278926 | [I-].CC[n+]1c(/C=C/Nc2ccccc2)sc2ccccc21
 5282075 | COC(=O)c1ccc2nc(Br)sc2c1
 5283354 | CCc1ccc2nc(N(C)CC(=O)O)sc2c1
 5283355 | Cc1ccc2nc(N(C)CC(=O)O)sc2c1
 5283356 | COc1ccc2nc(N(C)CC(=O)O)sc2c1
 5283357 | CCOc1ccc2nc(N(C)CC(=O)O)sc2c1
 ...
 4854425 | NC(=O)c1ccccc1NC(=O)C1CN(c2nc3c(cccc3F)s2)C1
(100 rows)

Time: 50.644 ms
```

6.2.4 SMARTS-based queries

Oxadiazole or thiadiazole:

```
emolecules=# select * from mols where m@>'c1[o,s]ncn1'::qmol limit 500;
   id    |                            m
---------+-----------------------------------------------------------
 5273135 | Cc1nsc(Br)n1
 5284275 | CCCC[Sn](CCCC)(CCCC)c1nc(C)ns1
 5192275 | CCCCCC(CC(=O)OCC)OC(=O)COCc1nc(C)no1
 5188130 | O=c1c2cccnc2ncn1Cc1nc(-c2ccoc2)no1
 5188272 | COCCc1noc(CNCC2CCCN2c2cccnn2)n1
 5188249 | Cc1nc(CN2CCCC(Nc3cc(C)nc4ncnn43)C2)no1
 5188283 | CN(Cc1nc(-c2ccco2)no1)CC1CCCN1c1cccnn1
```

```
 5188293 | COCCc1noc(CN(C)CC2CCCN2c2cccnn2)n1
 ...
 5037294 | Cc1noc(COc2cccc([N+](=O)[O-])c2C)n1
(500 rows)

Time: 313.202 ms
```

Notice that this is slower than the the pure SMILES query, this is generally true of SMARTS-based queries.

6.2.5 Similarity searches

Generating fingerprints and indices:

```
emolecules=# select id,torsionbv_fp(m) as torsionbv,morganbv_fp(m,2) as mfp2 into fps from mols;
SELECT 6008732
Time: 1734537.410 ms
emolecules=# create index mfp2idx on fps using gist(mfp2);
CREATE INDEX
Time: 381025.418 ms
emolecules=# create index torsionbvidx on fps using gist(torsionbv);
CREATE INDEX
Time: 379285.670 ms
emolecules=# alter table mols add primary key (id);
alter table fps add foreign key (id) references mols;NOTICE:  ALTER TABLE / ADD PRIMARY KEY will crea
ALTER TABLE
Time: 50798.813 ms
emolecules=# alter table fps add foreign key (id) references mols;
ALTER TABLE
Time: 39067.348 ms
```

Basic similarity searching:

```
emolecules=# select count(*) from fps where mfp2%morganbv_fp('Cc1ccc2nc(-c3ccc(NC(C4N(C(c5cccs5)=O)C
 count
-------
   513
(1 row)

Time: 4044.707 ms
```

Usually we'd like to find a sorted listed of neighbors along with the accompanying SMILES. This SQL function makes that pattern easy:

```
emolecules=# create or replace function get_mfp2_neighbors(smiles text)
  returns table(molregno int, m mol, similarity double precision) as
$$
select id,m,tanimoto_sml(morganbv_fp($1::mol),mfp2) as similarity
from fps join mols using (id)
where morganbv_fp($1::mol)%mfp2
order by morganbv_fp($1::mol)<%>mfp2;
$$ language sql stable ;
CREATE FUNCTION
Time: 600.371 ms
emolecules=# select * from get_mfp2_neighbors('Cc1ccc2nc(-c3ccc(NC(C4N(C(c5cccs5)=O)CCC4)=O)cc3)sc2c1
 molregno |                              m                               |    similarity
----------+--------------------------------------------------------------+------------------
  3116265 | Cc1ccc2nc(-c3ccc(NC(=O)[C@@H]4CCCN4C(=O)c4cccs4)cc3)sc2c1 |                1
  1598902 | Cc1ccc2sc(-c3ccc(NC(=O)C4CCCN4C(=O)c4cccs4)cc3)nc2c1      | 0.888888888888889
```

```
  3118194 | O=C(Nc1ccc(-c2nc3ccccc3s2)cc1)[C@@H]1CCCN1C(=O)c1cccs1    |          0.796875
  5695374 | Cc1ccc2nc(NC(=O)C3CCCN3C(=O)c3cccs3)sc2c1                  | 0.777777777777778
  1758570 | Cc1ccc2nc(-c3ccc(NC(=O)C4CCN(C(=O)c5cccs5)CC4)cc3)sc2c1    | 0.772727272727273
  4267350 | Cc1nc2ccc(NC(=O)[C@@H]3CCCN3C(=O)c3cccs3)cc2s1             | 0.738461538461539
  5825487 | Cc1ccc(NC(=O)C2CCCCN2C(=O)c2cccs2)cc1                      | 0.733333333333333
  2682124 | Cc1ccc2nc(-c3ccc(NC(=O)C4CCCN4S(C)(=O)=O)cc3)sc2c1         | 0.701492537313433
  3552075 | Cc1ccc2nc(-c3ccc(NC(=O)C4CCCCN4S(C)(=O)=O)cc3)sc2c1        | 0.686567164179104
  1807011 | CSc1nc2ccc(NC(=O)C3CCCN3C(=O)c3cccs3)cc2s1                 | 0.671428571428571
(10 rows)

Time: 4156.841 ms
emolecules=# select * from get_mfp2_neighbors('Cc1ccc2nc(N(C)CC(=O)O)sc2c1') limit 10;
 molregno |                   m                    |    similarity
----------+----------------------------------------+------------------
  5283355 | Cc1ccc2nc(N(C)CC(=O)O)sc2c1            |                1
  5283354 | CCc1ccc2nc(N(C)CC(=O)O)sc2c1           | 0.761904761904762
  5283360 | CN(CC(=O)O)c1nc2ccc(Br)cc2s1           |  0.75609756097561
  5283363 | CN(CC(=O)O)c1nc2ccc(F)cc2s1            | 0.738095238095238
  5283369 | CN(CC(=O)O)c1nc2ccc(Cl)cc2s1           | 0.738095238095238
  5283365 | Cc1cc2nc(N(C)CC(=O)O)sc2cc1C           |             0.725
  5283367 | CN(CC(=O)O)c1nc2ccc(S(C)(=O)=O)cc2s1   | 0.720930232558139
  5283356 | COc1ccc2nc(N(C)CC(=O)O)sc2c1           | 0.704545454545455
  5283362 | CC(C)c1ccc2nc(N(C)CC(=O)O)sc2c1        | 0.704545454545455
  5283358 | CSc1ccc2nc(N(C)CC(=O)O)sc2c1           | 0.704545454545455
(10 rows)

Time: 4186.420 ms
```

6.3 Reference Guide

6.3.1 New Types

- *mol* : an rdkit molecule. Can be created from a SMILES via direct type conversion, for example: *'c1ccccc1'::mol* creates a molecule from the SMILES *'c1ccccc1'*
- *qmol* : an rdkit molecule containing query features (i.e. constructed from SMARTS). Can be created from a SMARTS via direct type conversion, for example: *'c1cccc[c,n]1'::qmol* creates a query molecule from the SMARTS *'c1cccc[c,n]1'*
- *sfp* : a sparse count vector fingerprint (*SparseIntVect* in C++ and Python)
- *bfp* : a bit vector fingerprint (*ExplicitBitVect* in C++ and Python)

6.3.2 Parameters

- *rdkit.tanimoto_threshold* : threshold value for the Tanimoto similarity operator. Searches done using Tanimoto similarity will only return results with a similarity of at least this value.
- *rdkit.dice_threshold* : threshold value for the Dice similiarty operator. Searches done using Dice similarity will only return results with a similarity of at least this value.

6.3.3 Operators

Similarity search

- % : operator used for similarity searches using Tanimoto similarity. Returns whether or not the Tanimoto similarity between two fingerprints (either two *sfp* or two *bfp* values) exceeds *rdkit.tanimoto_threshold*.
- # : operator used for similarity searches using Dice similarity. Returns whether or not the Dice similarity between two fingerprints (either two *sfp* or two *bfp* values) exceeds *rdkit.dice_threshold*.
- <%> : used for Tanimoto KNN searches (to return ordered lists of neighbors).
- <#> : used for Dice KNN searches (to return ordered lists of neighbors).

Substructure and exact structure search

- @> : substructure search operator. Returns whether or not the *mol* or *qmol* on the right is a substructure of the *mol* on the left.
- <@ : substructure search operator. Returns whether or not the *mol* or *qmol* on the left is a substructure of the *mol* on the right.
- @= : returns whether or not two molecules are the same.

Molecule comparison

- < : returns whether or not the left *mol* is less than the right *mol*
- > : returns whether or not the left *mol* is greater than the right *mol*
- = : returns whether or not the left *mol* is equal to the right *mol*
- <= : returns whether or not the left *mol* is less than or equal to the right *mol*
- >= : returns whether or not the left *mol* is greater than or equal to the right *mol*

Note Two molecules are compared by making the following comparisons in order. Later comparisons are only made if the preceding values are equal:

Number of atoms # Number of bonds # Molecular weight # Number of rings

If all of the above are the same and the second molecule is a substructure of the first, the molecules are declared equal, Otherwise (should not happen) the first molecule is arbitrarily defined to be less than the second.

There are additional operators defined in the cartridge, but these are used for internal purposes.

6.3.4 Functions

Fingerprint Related

Generating fingerprints

- *morgan_fp(mol,int)* : returns an *sfp* which is the count-based Morgan fingerprint for a molecule using connectivity invariants. The second argument provides the radius. This is an ECFP-like fingerprint.
- *morganbv_fp(mol,int)* : returns a *bfp* which is the bit vector Morgan fingerprint for a molecule using connectivity invariants. The second argument provides the radius. This is an ECFP-like fingerprint.

- *featmorgan_fp(mol,int)* : returns an *sfp* which is the count-based Morgan fingerprint for a molecule using chemical-feature invariants. The second argument provides the radius. This is an FCFP-like fingerprint.
- *featmorganbv_fp(mol,int)* : returns a *bfp* which is the bit vector Morgan fingerprint for a molecule using chemical-feature invariants. The second argument provides the radius. This is an FCFP-like fingerprint.
- *rdkit_fp(mol)* : returns a *bfp* which is the RDKit fingerprint for a molecule. This is a daylight-fingerprint using hashed molecular subgraphs.
- *atompair_fp(mol)* : returns an *sfp* which is the count-based atom-pair fingerprint for a molecule.
- *atompairbv_fp(mol)* : returns a *bfp* which is the bit vector atom-pair fingerprint for a molecule.
- *torsion_fp(mol)* : returns an *sfp* which is the count-based topological-torsion fingerprint for a molecule.
- *torsionbv_fp(mol)* : returns a *bfp* which is the bit vector topological-torsion fingerprint for a molecule.
- *layered_fp(mol)* : returns a *bfp* which is the layered fingerprint for a molecule. This is an experimental substructure fingerprint using hashed molecular subgraphs.
- *maccs_fp(mol)* : returns a *bfp* which is the MACCS fingerpring for a molecule (*available from 2013_01 release*).

Working with fingerprints

- *tanimoto_sml(fp,fp)* : returns the Tanimoto similarity between two fingerprints of the same type (either two *sfp* or two *bfp* values).
- *dice_sml(fp,fp)* : returns the Dice similarity between two fingerprints of the same type (either two *sfp* or two *bfp* values).
- *size(bfp)* : returns the length of (number of bits in) a *bfp*.
- *add(sfp,sfp)* : returns an *sfp* formed by the element-wise addition of the two *sfp* arguments.
- *subtract(sfp,sfp)* : returns an *sfp* formed by the element-wise subtraction of the two *sfp* arguments.
- *all_values_lt(sfp,int)* : returns a boolean indicating whether or not all elements of the *sfp* argument are less than the *int* argument.
- *all_values_gt(sfp,int)* : returns a boolean indicating whether or not all elements of the *sfp* argument are greater than the *int* argument.

Fingerprint I/O

- *bfp_to_binary_text(bfp)* : returns a bytea with the binary string representation of the fingerprint that can be converted back into an RDKit fingerprint in other software. (*available from Q3 2012 (2012_09) release*)
- *bfp_from_binary_text(bytea)* : constructs a bfp from a binary string representation of the fingerprint. (*available from Q3 2012 (2012_09) release*)

Molecule Related

Molecule I/O and Validation

- *is_valid_smiles(smiles)* : returns whether or not a SMILES string produces a valid RDKit molecule.
- *is_valid_ctab(ctab)* : returns whether or not a CTAB (mol block) string produces a valid RDKit molecule.
- *is_valid_smarts(smarts)* : returns whether or not a SMARTS string produces a valid RDKit molecule.

- *is_valid_mol_pkl(bytea)* : returns whether or not a binary string (bytea) can be converted into an RDKit molecule. (*available from Q3 2012 (2012_09) release*)
- *mol_from_smiles(smiles)* : returns a molecule for a SMILES string, NULL if the molecule construction fails.
- *mol_from_smarts(smarts)* : returns a molecule for a SMARTS string, NULL if the molecule construction fails.
- *mol_from_ctab(ctab)* : returns a molecule for a CTAB (mol block) string, NULL if the molecule construction fails.
- *mol_from_pkl(bytea)* : returns a molecule for a binary string (bytea), NULL if the molecule construction fails. (*available from Q3 2012 (2012_09) release*)
- *mol_to_smiles(mol)* : returns the canonical SMILES for a molecule.
- *mol_to_smarts(mol)* : returns SMARTS string for a molecule.
- *mol_to_pkl(mol)* : returns binary string (bytea) for a molecule. (*available from Q3 2012 (2012_09) release*)

Descriptors

- *mol_amw(mol)* : returns the AMW for a molecule.
- *mol_logp(mol)* : returns the MolLogP for a molecule.
- *mol_tpsa(mol)* : returns the topological polar surface area for a molecule (*available from Q1 2011 (2011_03) release*).
- *mol_fractioncsp3(mol)* : returns the fraction of carbons that are sp3 hybridized (*available from 2013_03 release*).
- *mol_hba(mol)* : returns the number of Lipinski H-bond acceptors (i.e. number of Os and Ns) for a molecule.
- *mol_hbd(mol)* : returns the number of Lipinski H-bond donors (i.e. number of Os and Ns that have at least one H) for a molecule.
- *mol_numatoms(mol)* : returns the total number of atoms in a molecule.
- *mol_numheavyatoms(mol)* : returns the number of heavy atoms in a molecule.
- *mol_numrotatablebonds(mol)* : returns the number of rotatable bonds in a molecule (*available from Q1 2011 (2011_03) release*).
- *mol_numheteroatoms(mol)* : returns the number of heteroatoms in a molecule (*available from Q1 2011 (2011_03) release*).
- *mol_numrings(mol)* : returns the number of rings in a molecule (*available from Q1 2011 (2011_03) release*).
- *mol_numaromaticrings(mol)* : returns the number of aromatic rings in a molecule (*available from 2013_03 release*).
- *mol_numaliphaticrings(mol)* : returns the number of aliphatic (at least one non-aromatic bond) rings in a molecule (*available from 2013_03 release*).
- *mol_numsaturatedrings(mol)* : returns the number of saturated rings in a molecule (*available from 2013_03 release*).
- *mol_numaromaticheterocycles(mol)* : returns the number of aromatic heterocycles in a molecule (*available from 2013_03 release*).
- *mol_numaliphaticheterocycles(mol)* : returns the number of aliphatic (at least one non-aromatic bond) heterocycles in a molecule (*available from 2013_03 release*).
- *mol_numsaturatedheterocycles(mol)* : returns the number of saturated heterocycles in a molecule (*available from 2013_03 release*).

- *mol_numaromaticcarbocycles(mol)* : returns the number of aromatic carbocycles in a molecule (*available from 2013_03 release*).
- *mol_numaliphaticcarbocycles(mol)* : returns the number of aliphatic (at least one non-aromatic bond) carbocycles in a molecule (*available from 2013_03 release*).
- *mol_numsaturatedcarbocycles(mol)* : returns the number of saturated carbocycles in a molecule (*available from 2013_03 release*).
- *mol_inchi(mol)* : returns an InChI for the molecule. (*available from the 2011_06 release, requires that the RDKit be built with InChI support*).
- *mol_inchikey(mol)* : returns an InChI key for the molecule. (*available from the 2011_06 release, requires that the RDKit be built with InChI support*).

Connectivity Descriptors

- *mol_chi0v(mol)* - *mol_chi4v(mol)* : returns the ChiXv value for a molecule for X=0-4 (*available from 2012_01 release*).
- *mol_chi0n(mol)* - *mol_chi4n(mol)* : returns the ChiXn value for a molecule for X=0-4 (*available from 2012_01 release*).
- *mol_kappa1(mol)* - *mol_kappa3(mol)* : returns the kappaX value for a molecule for X=1-3 (*available from 2012_01 release*).

Other

- *rdkit_version()* : returns a string with the cartridge version number.

There are additional functions defined in the cartridge, but these are used for internal purposes.

6.4 License

CHAPTER

SEVEN

ADDITIONAL INFORMATION

- Python API Documentation
- C++ API Documentation