RDKit Documentation

Release 2017.09.1.dev1

Greg Landrum

Apr 26, 2017

Contents

CHAPTER 1

An overview of the RDKit

What is it?

Open source toolkit for cheminformatics

- Business-friendly BSD license
- Core data structures and algorithms in C++
- Python (2.x and 3.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
- Releases every 6 months
- Web presence:
  - Homepage: http://www.rdkit.org Documentation, links
  - Github (https://github.com/rdkit) Downloads, bug tracker, git repository
  - Sourceforge (http://sourceforge.net/projects/rdkit) Mailing lists

  - Blog (https://rdkit.blogspot.com) Tips, tricks, random stuff
  - Tutorials (https://github.com/rdkit/rdkit-tutorials) Jupyter-based tutorials for using the RDKit
  - KNIME integration (https://github.com/rdkit/knime-rdkit) RDKit nodes for KNIME
- Mailing lists at https://sourceforge.net/p/rdkit/mailman/, searchable archives available for rdkit-discuss and rdkit-devel
- Social media:
  - Twitter: @RDKit_org
  - LinkedIn: https://www.linkedin.com/groups/8192558
  - Google+: https://plus.google.com/u/0/116996224395614252219
  - Slack: https://rdkit.slack.com (invite required, contact Greg)

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

Functionality overview

Basics

- Input/Output: SMILES/SMARTS, SDF, TDT, SLN *1*, Corina mol2 *1*, PDB, sequence notation, FASTA (peptides only), HELM (peptides only)
- 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
- Fingerprinting: Daylight-like, atom pairs, topological torsions, Morgan algorithm, “MACCS keys”, extended reduced graphs, etc.
- Similarity/diversity picking
- Gasteiger-Marsili charges
- Bemis and Murcko scaffold determination
- Salt stripping
- Functional-group filters

2D

- 2D pharmacophores *1*
- Hierarchical subgraph/fragment analysis
- RECAP and BRICS implementations
- Multi-molecule maximum common substructure *2*
- Enumeration of molecular resonance structures
- 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
  - MQN *6*
- Similarity Maps *7*
- Machine Learning:
  - Clustering (hierarchical, Butina)
  - Information theory (Shannon entropy, information gain, etc.)
- Tight integration with the Jupyter notebook (formerly the IPython notebook) and Pandas.

3D

- 2D->3D conversion/conformational analysis via distance geometry, including optional use of experimental torsion angle potentials *9*
- UFF and MMFF94/MMFF94S implementations for cleaning up structures
- Pharmacophore embedding (generate a pose of a molecule that matches a 3D pharmacophore) *1*
- Feature maps
- Shape-based similarity
- RMSD-based molecule-molecule alignment
- Shape-based alignment (subshape alignment *3*) *1*
- Unsupervised molecule-molecule alignment using the Open3DAlign algorithm *4*
- Integration with PyMOL for 3D visualization
- Molecular descriptor library:
  - Moments-of-inertia based descriptors: PMI, NPR, PBF, etc.
  - Feature-map vectors *5*
- Torsion Fingerprint Differences for comparing conformations *8*

Integration with other open-source projects

- KNIME: Workflow and analytics tool
- Django: "The web framework for perfectionists with deadlines"
- PostgreSQL: Extensible relational database
- Lucene: Text-search engine *1*

Usage by other open-source projects

- ChEMBL Beaker - standalone web server wrapper for RDKit and OSRA
- myChEMBL (blog post, paper) - A virtual machine implementation of open data and cheminformatics tools
- ZINC - Free database of commercially-available compounds for virtual screening
- sdf_viewer.py - an interactive SDF viewer
- sdf2ppt - Reads an SDFile and displays molecules as image grid in powerpoint/openoffice presentation.
- MolGears - A cheminformatics tool for bioactive molecules
- PYPL - Simple cartridge that lets you call Python scripts from Oracle PL/SQL.
- shape-it-rdkit - Gaussian molecular overlap code shape-it (from silicos it) ported to RDKit backend
- WONKA - Tool for analysis and interrogation of protein-ligand crystal structures
- OOMMPPAA - Tool for directed synthesis and data analysis based on protein-ligand crystal structures
- OCEAN - web-tool for target-prediction of chemical structures which uses ChEMBL as datasource
- chemfp - very fast fingerprint searching
- rdkit_ipynb_tools - RDKit Tools for the IPython Notebook
- chemicalite - SQLite integration for the RDKit
- Vernalis KNIME nodes
- Erlwood KNIME nodes
- AZOrange

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:

- A. 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

- 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

Contribution from Nicholas Firth

Note as of the 2016.09.1 release this functionality is part of the RDKit core.

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

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

mmpa: Matched molecular pairs

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

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.

SA_Score: Synthetic assessibility score

Python source for an implementation of the SA score algorithm described in the publication:

Ertl, P. and Schuffenhauer A. "Estimation of Synthetic Accessibility Score of Drug-like Molecules based on Molecular Complexity and Fragment Contributions" *Journal of Cheminformatics* **1:8** (2009)

Contribution from Peter Ertl

fraggle: A fragment-based molecular similarity algorithm

Python source for an implementation of the fraggle similarity algorithm developed at GSK and described in this RDKit UGM presentation: https://github.com/rdkit/UGM_2013/blob/master/Presentations/Hussain.Fraggle.pdf

Contribution from Jameed Hussain

pzc: Tools for building and validating classifiers

Contribution from Paul Czodrowski

ConformerParser: parser for Amber trajectory files

Contribution from Sereina Riniker

Note as of the 2016.09.1 release this functionality is part of the RDKit core.

NP_Score: Natural-product likeness score

Python source for an implementation of the NP score algorithm described in the publication:

"Natural Product Likeness Score and Its Application for Prioritization of Compound Libraries" Peter Ertl, Silvio Roggo, and Ansgar Schuffenhauer *Journal of Chemical Information and Modeling* **48:68-74** (2008) http://pubs.acs.org/doi/abs/10.1021/ci700286x

Contribution from Peter Ertl

AtomAtomSimilarity: atom-atom-path method for fragment similarity

Python source for an implementation of the Atom-Atom-Path similarity method for fragments described in the publication:

Gobbi, A., Giannetti, A. M., Chen, H. & Lee, M.-L. "Atom-Atom-Path similarity and Sphere Exclusion clustering: tools for prioritizing fragment hits." *J. Cheminformatics* **7** 11 (2015). http://dx.doi.org10.1186/s13321-015-0056-8

Contribution from Richard Hall

Footnotes

1: These implementations are functional but are not necessarily the best, fastest, or most complete.

2: Originally contributed by 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).

4: Tosco, P., Balle, T. & Shiri, F. "Open3DALIGN: an open-source software aimed at unsupervised ligand alignment." *J Comput Aided Mol Des* **25:777–83** (2011).

5: 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).

6: Nguyen, K. T., Blum, L. C., van Deursen, R. & Reymond, J.-L. "Classification of Organic Molecules by Molecular Quantum Numbers." *ChemMedChem* **4:1803–5** (2009).

7: Riniker, S. & Landrum, G. A. "Similarity maps - a visualization strategy for molecular fingerprints and machine-learning methods." *Journal of Cheminformatics* **5:43** (2013).

8: Schulz-Gasch, T., Schärfer, C., Guba, W. & Rarey, M. "TFD: Torsion Fingerprints As a New Measure To Compare Small Molecule Conformations." *J. Chem. Inf. Model.* **52:1499–1512** (2012).

9: Riniker, S. & Landrum, G. A. "Better informed distance geometry: Using what we know to improve conformation generation." *J. Chem. Inf. Model.* **55:2562–74** (2015).

License

The intent of this license is similar to that of the RDKit itself. In simple words: "Do whatever you want with it, but please give us some credit."

CHAPTER 2

Installation

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

Cross-platform under anaconda python (fastest install)

Introduction to anaconda

Conda is an open-source, cross-platform, software package manager. It supports the packaging and distribution of software components, and manages their installation inside isolated execution environments. It has several analogies with pip and virtualenv, but it is designed to be more "python-agnostic" and more suitable for the distribution of binary packages and their dependencies.

How to get conda

The easiest way to get Conda is having it installed as part of the Anaconda Python distribution. A possible (but a bit more complex to use) alternative is provided with the smaller and more self-contained Miniconda. The conda source code repository is available on github and additional documentation is provided by the project website.

How to install RDKit with Conda

Creating a new conda environment with the RDKit installed using these packages requires one single command similar to the following::

$ conda create -c rdkit -n my-rdkit-env rdkit

Finally, the new environment must be activated, so that the corresponding python interpreter becomes available in the same shell:

$ source activate my-rdkit-env

If for some reason this does not work, try:

$ cd [anaconda folder]/bin $ source activate my-rdkit-env

Windows users will use a slightly different command:

C:> activate my-rdkit-env

How to build from source with Conda

For more details on building from source with Conda, see the conda-rdkit repository.

macOS 10.12 (Sierra): Python 3 environment

The following commands will create a development environment for macOS Sierra and Python 3. Download Miniconda3-latest-MacOSX-x86_64.sh from Conda and run these following commands:

```
bash Miniconda3-latest-MacOSX-x86_64.sh
conda install numpy matplotlib
conda install cmake
conda install --channel rdkit boost
conda install --channel rdkit nox
conda install --channel rdkit cairo
conda install pillow
conda install anaconda
conda install --channel conda-forge eigen
conda install --channel conda-forge pkg-config
```

Optionally, add the following packages to your environment as useful development tools.

```
pip install yapf==0.11.1
pip install coverage==3.7.1
```

Then follow the usual build instructions. The PYTHON_INCLUDE_DIR must be set in the cmake command.

```
PYROOT=<path to miniconda3>
cmake -DPYTHON_INCLUDE_DIR=$PYROOT/include/python3.5m  \
  -DRDK_BUILD_AVALON_SUPPORT=ON \
  -DRDK_BUILD_CAIRO_SUPPORT=ON \
  -DRDK_BUILD_INCHI_SUPPORT=ON \
  ..
```

Once “make” and “make install” completed successfully, use the following command to run the tests:

```
RDBASE=$RDBASE DYLD_FALLBACK_LIBRARY_PATH="$RDBASE/lib:$PYROOT/lib" PYTHONPATH=
↪$RDBASE ctest
```

This is required due to the System Integrity Protection SIP introduced in more recent macOS versions.

Installing and using PostgreSQL and the RDKit PostgreSQL cartridge from a conda environment

Due to the conda python distribution being a different version to the system python, it is easiest to install PostgreSQL and the PostgreSQL python client via conda.

With your environment activated, this is done simply by:

```
conda install -c rdkit rdkit-postgresql
```

The conda packages PostgreSQL version needs to be initialized by running the initdb command found in [conda folder]/envs/my-rdkit-env/bin

```
[conda folder]/envs/my-rdkit-env/bin/initdb -D /folder/where/data/should/be/stored
```

PostgreSQL can then be run from the terminal with the command:

```
[conda folder]/envs/my-rdkit-env/bin/postgres -D /folder/where/data/should/be/stored
```

For most use cases you will instead need to run PostgreSQL as a daemon, one way to do this is using supervisor. You can find out more and how to install supervisor here. The required configuration file will look something like this:

```
[program:postgresql]
command=[conda folder]/envs/my-rdkit-env/bin/postgres -D /folder/where/data/should/be/
↪stored
user=[your username]
autorestart=true
```

Once PostgreSQL is up and running, all of the normal PostgreSQL commands can then be run when your conda environment is activated. Therefore to create a database you can run:

```
createdb my_rdkit_db
psql my_rdkit_db
# create extension rdkit;
```

If you are trying to use multiple installations of PostgreSQL in different environments, you will need to setup different pid files, unix sockets and ports by editing the PostgreSQL config files. With the above configurations these files can be found in /folder/where/data/should/be/stored.

Linux and OS X

Installation from repositories

Ubuntu 12.04 and later

Thanks to the efforts of the Debichem team, RDKit is available via the Ubuntu repositories. To install:

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

Fedora, CentOS, and RHEL

Thanks to Gianluca Sforna's work, binary RPMs for the RDKit are now part of the official Fedora repositories: https://admin.fedoraproject.org/pkgdb/package/rpms/rdkit/

OS X

Eddie Cao has produced a homebrew formula that can be used to easily build the RDKit https://github.com/rdkit/homebrew-rdkit

Building from Source

Prerequisites

Installing prerequisites as packages

Ubuntu and other debian-derived systems

Install the following packages using apt-get:

```
build-essential python-numpy cmake python-dev sqlite3 libsqlite3-dev libboost-dev␣
↪libboost-system-dev libboost-thread-dev libboost-serialization-dev libboost-python-
↪dev libboost-regex-dev
```

Fedora, CentOS (5.7+), and RHEL

Install the following packages using yum:

```
cmake tk-devel readline-devel zlib-devel bzip2-devel sqlite-devel @development-tools
```

Packages to install from source (not required on RHEL/CentOS 6.x):

- python 2.7 : use `./configure CFLAGS=-fPIC --enable-unicode=ucs4 --enable-shared`
- numpy : do `export LD\_LIBRARY\_PATH="/usr/local/lib"` before `python setup.py install`
- boost 1.48.0 or later: do `./bootstrap.sh --with-libraries=python,regex; ./b2; ./b2 install`

Older versions of CentOS

Here things are more difficult. Check this wiki page for information: https://code.google.com/p/rdkit/wiki/BuildingOnCentOS

Installing prerequisites from source

- 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.

- 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 OS X there seems to be a problem with the version of numpy that comes with XCode4. Please see below in the (see faq) section for a workaround.

Installing Boost

If your linux distribution has a boost-devel package including the python, regex, threading, and serialization 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 FAQ) 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_58_0`)
- build the required boost libraries. The boost site has detailed instructions for this, but here's an overview:
- `cd $BOOST`
- If you want to use the python wrappers: `./bootstrap.sh --with-libraries=python,regex,thread,serialization`
- If not using the python wrappers: `./bootstrap.sh --with-libraries=regex,thread,serialization`
- `./b2 install`

```
If you have any problems with this step, check the boost [installation␣
↪instructions](http://www.boost.org/more/getting_started/unix-variants.html).
```

Building the RDKit

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

```
wget https://github.com/rdkit/rdkit/archive/Release_XXXX_XX_X.tar.gz
```

- Ensure that the prerequisites are installed
- 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
  - *OS X:* 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 FAQ and solutions.

Testing the build (optional, but recommended)

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

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 ..
```

Note that if you are using your own boost install on a system with a system install, it's normally a good idea to also include the argument `-D Boost_NO_SYSTEM_PATHS=ON` in your cmake command.

Specifying an alternate Python installation

If you aren't using the default python installation for your computer, 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/python2.7/config/libpython2.7.a -D PYTHON_INCLUDE_
↪DIR=/usr/include/python2.7/ -D PYTHON_EXECUTABLE=/usr/bin/python ..
```

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:

```
cmake -DRDK_BUILD_PYTHON_WRAPPERS=OFF ..
```

Recommended extras

- You can enable support for generating InChI strings and InChI keys by adding the argument `-DRDK_BUILD_INCHI_SUPPORT=ON` to your cmake command line.
- You can enable support for the Avalon toolkit by adding the argument `-DRDK_BUILD_AVALON_SUPPORT=ON` to your cmake command line.
- If you'd like to be able to generate high-quality PNGs for structure depiction cairo (for use with Python2) or cairocffi (for use with Python3) and their respective Python bindings are recommended.

Building the Java wrappers

Additional Requirements

- SWIG v2.0.x: http://www.swig.org

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 OS X), 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=$RDBASE/Code/JavaWrappers/gmwrapper
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
```

Optional packages

- If you would like to install the RDKit InChI support, follow the instructions in `$RDBASE/External/INCHI-API/README`.
- If you would like to install the RDKit Avalon toolkit support, follow the instructions in `$RDBASE/External/AvalonTool/README`.
- If you would like to build and install the PostgreSQL cartridge, follow the instructions in `$RDBASE/Code/PgSQL/rdkit/README`.

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_string<char, std::char_traits<char>, std::allocator<char> >::_
↪Rep::_S_empty_rep_storage' can not be used when making a shared object; recompile␣
↪with -fPIC
.../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: http://www.mail-archive.com/rdkit-discuss@lists.sourceforge.net/msg01119.html

---

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_nonlocal_decl, at cp/decl.c:5067

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: http://www.mail-archive.com/rdkit-discuss@lists.sourceforge.net/msg01178.html

---

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 OS X 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.

Windows

Prerequisites

- Python 2.7 or 3.4+ (from http://www.python.org/)
- numpy (from http://numpy.scipy.org/ or use `pip install numpy`). Binaries for win64 are available here: http://www.lfd.uci.edu/~gohlke/pythonlibs/#numpy
- Pillow: (from https://python-pillow.github.io/> or use `pip install Pillow`)

Recommended extras

- aggdraw: a library for high-quality drawing in Python. Instructions for downloading are here: http://effbot.org/zone/aggdraw-index.htm The new (as of May 2008) drawing code has been tested with v1.2a3 of aggdraw. Despite the alpha label, the code is stable and functional.
- matplotlib: a library for scientific plotting from Python. http://matplotlib.sourceforge.net/
- ipython : a very useful interactive shell (and much more) for Python. http://ipython.scipy.org/dist/
- win32all: Windows extensions for Python. http://sourceforge.net/projects/pywin32/

Installation of RDKit binaries

- Get the appropriate windows binary build from: https://github.com/rdkit/rdkit/releases
- Extract the zip file somewhere without a space in the name, i.e. `C:\`
- The rest of this will assume that the installation is in `C:\RDKit_2015_09_2`
- Set the following environment variables:
- RDBASE: `C:\RDKit_2015_09_2`
- PYTHONPATH: `%RDBASE%` if there is already a PYTHONPATH, put `;%RDBASE%` at the end.

- PATH: add `;%RDBASE%\lib` to the end

In Win7 systems, you may run into trouble due to missing DLLs, see one thread from the mailing list: http://www.mail-archive.com/rdkit-discuss@lists.sourceforge.net/msg01632.html You can download the missing DLLs from here: http://www.microsoft.com/en-us/download/details.aspx?id=5555

Installation from source

Extra software to install

- Microsoft Visual C++ : The Community version has everything necessary and can be downloaded for free (https://www.visualstudio.com/en-us/products/visual-studio-community-vs.aspx). This is a big installation and will take a while. The RDKit has been successfully built with all version of Visual C++ since 6.0, so the current version of VC++ (2015 as of this writing) should be fine.
- cmake : (http://www.cmake.org/cmake/resources/software.html) should be installed.
- boost : It is strongly recommended to download and use a precompiled version of the boost libraries from http://sourceforge.net/projects/boost/files/boost-binaries/ . When you run the installer, the only binary libraries you need are python, regex, and system. If you want to install boost from source, download a copy from http://www.boost.org and follow the instructions in the "Getting Started" section of the documentation. Make sure the libraries and headers are installed to C:\boost
- a git client : *This is only necessary if you are planning on building development versions of the RDKit.* This can be downloaded from http://git-scm.com/downloads; git is also included as an optional add-on of Microsoft Visual Studio 2015.

Setup and Preparation

This section assumes that python is installed in `C:\Python27`, that the boost libraries have been installed to `C:\boost`, and that you will build the RDKit from a directory named `C:\RDKit`. If any of these conditions is not true, just change the corresponding paths.

- If you install things in paths that have spaces in their names, be sure to use quotes properly in your environment variable definitions.
- If you are planning on using a development version of the RDKit: get a copy of the current RDKit source using git. If you're using the command-line client the command is: `git clone https://github.com/rdkit/rdkit.git C:\RDKit`
- If you are planning on using a released version of the RDKit: get a copy of the most recent release and extract it into the directory `C:\RDKit`
- Set the required environment variables:
- `RDBASE = C:\RDKit`
- Make sure `C:\Python27` is in your PATH
- Make sure `C:\RDKit\lib` is in your PATH
- Make sure `C:\boost\lib` is in your PATH.
- Make sure `C:\RDKit is` in your PYTHONPATH

Building from the command line (recommended)

- Create a directory `C:\RDKit\build` and cd into it
- Run cmake. Here's an example basic command line for 64bit windows that will download the InChI and Avalon toolkit sources from the InChI Trust and SourceForge repositories, respectively, and build the PostgreSQL cartridge for the installed version of PostgreSQL: `cmake -DRDK_BUILD_PYTHON_WRAPPERS=ON -DBOOST_ROOT=C:/boost -DRDK_BUILD_INCHI_SUPPORT=ON -DRDK_BUILD_AVALON_SUPPORT=ON -DRDK_BUILD_PGSQL=ON -DPostgreSQL_ROOT="C:\Program Files\PostgreSQL\9.5" -G"Visual Studio 14 2015 Win64" ..`
- Build the code. Here's an example command line: `C:/Windows/Microsoft.NET/Framework64/v4.0.30319/MSBuild.exe /m:4 /p:Configuration=Release INSTALL.vcxproj`
- If you have built in PostgreSQL support, you will need to open a shell with administrator privileges, stop the PostgreSQL service, run the `pgsql_install.bat` installation script, then restart the PostgreSQL service (please refer to `%RDBASE%\Code\PgSQL\rdkit\README` for further details):
  - `"C:\Program Files\PostgreSQL\9.5\bin\pg_ctl.exe" -N "postgresql-9.5" -D "C:\Program Files\PostgreSQL\9.5\data" -w stop`
  - `C:\RDKit\build\Code\PgSQL\rdkit\pgsql_install.bat`
  - `"C:\Program Files\PostgreSQL\9.5\bin\pg_ctl.exe" -N "postgresql-9.5" -D "C:\Program Files\PostgreSQL\9.5\data" -w start`
  - Before restarting the PostgreSQL service, make sure that the Boost libraries the RDKit was built against are in the system PATH, or PostgreSQL will fail to create the `rdkit` extension with a deceptive error message such as: `ERROR: could not load library "C:/Program Files/PostgreSQL/9.5/lib/rdkit.dll": The specified module could not be found.`

Testing the Build (optional, but recommended)

- cd to `C:\RDKit\build` and run ctest. Please note that if you have built in PostgreSQL support, the current logged in user needs to be a PostgreSQL user with database creation and superuser privileges, or the PostgreSQL test will fail. A convenient option to authenticate will be to set the `PGPASSWORD` environment variable to the PostgreSQL password of the current logged in user in the shell from which you are running ctest.
- You're done!

License

The intent of this license is similar to that of the RDKit itself. In simple words: "Do whatever you want with it, but please give us some credit."

CHAPTER 3

Getting Started with the RDKit in Python

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 In particular, if you find yourself spending time working out how to do something that doesn't appear to be documented please contribute by writing it up for this document. Contributing to the documentation is a great service both to the RDKit community and to your future self.

Reading and Writing Molecules

Reading single molecules

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

```
>>> from __future__ import print_function
>>> 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=open('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
```

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 = open('data/5ht3ligs.sdf','rb')
>>> 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
```

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
```

In order for atom or bond stereochemistry to be recognised correctly by most software, it's essential that the Mol block have atomic coordinates. It's also convenient for many reasons, such as drawing the molecules. Coordinates 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 (we're using the ETKDG method here, which is described in more detail below):

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

  4  4  0  0  0  0  0  0  0  0999 V2000
   -0.8321    0.5405   -0.1981 C   0  0  0  0  0  0  0  0  0  0  0  0
   -0.3467   -0.8825   -0.2651 C   0  0  0  0  0  0  0  0  0  0  0  0
    0.7190   -0.5613    0.7314 C   0  0  0  0  0  0  0  0  0  0  0  0
    0.4599    0.9032    0.5020 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 get good 3D conformations, it's almost always a good idea to add hydrogens to the molecule first:

```
>>> m3 = Chem.AddHs(m2)
>>> AllChem.EmbedMolecule(m3,AllChem.ETKDG())
0
```

These can then be removed:

```
>>> m3 = Chem.RemoveHs(m3)
>>> print(Chem.MolToMolBlock(m3))
cyclobutane
     RDKit          3D

  4  4  0  0  0  0  0  0  0  0999 V2000
    0.3497    0.9755   -0.2202 C   0  0  0  0  0  0  0  0  0  0  0  0
    0.9814   -0.3380    0.2534 C   0  0  0  0  0  0  0  0  0  0  0  0
   -0.3384   -1.0009   -0.1474 C   0  0  0  0  0  0  0  0  0  0  0  0
   -0.9992    0.3532    0.1458 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
```

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

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

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 rdkit.six 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  1  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`.

Working with Molecules

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(atom.GetNeighbors()[-1].GetBonds())
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
```

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
```

By default, the bonds are still marked as being aromatic:

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

because the flags in the original molecule are not cleared (clearAromaticFlags defaults to False). You can explicitly force or decline a clearing of the flags:

```
>>> m = Chem.MolFromSmiles('c1ccccc1')
>>> m.GetBondWithIdx(0).GetIsAromatic()
True
>>> m1 = Chem.MolFromSmiles('c1ccccc1')
>>> Chem.Kekulize(m1, clearAromaticFlags=True)
>>> m1.GetBondWithIdx(0).GetIsAromatic()
False
```

Bonds 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.

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:

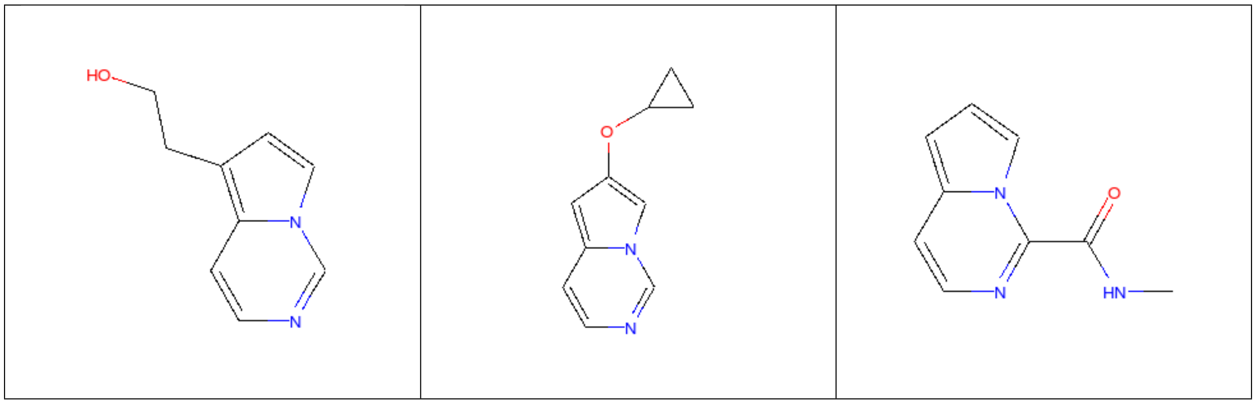
```
>>> m = Chem.MolFromSmiles('c1nccc2n1ccc2')
>>> AllChem.Compute2DCoords(m)
0
```

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:

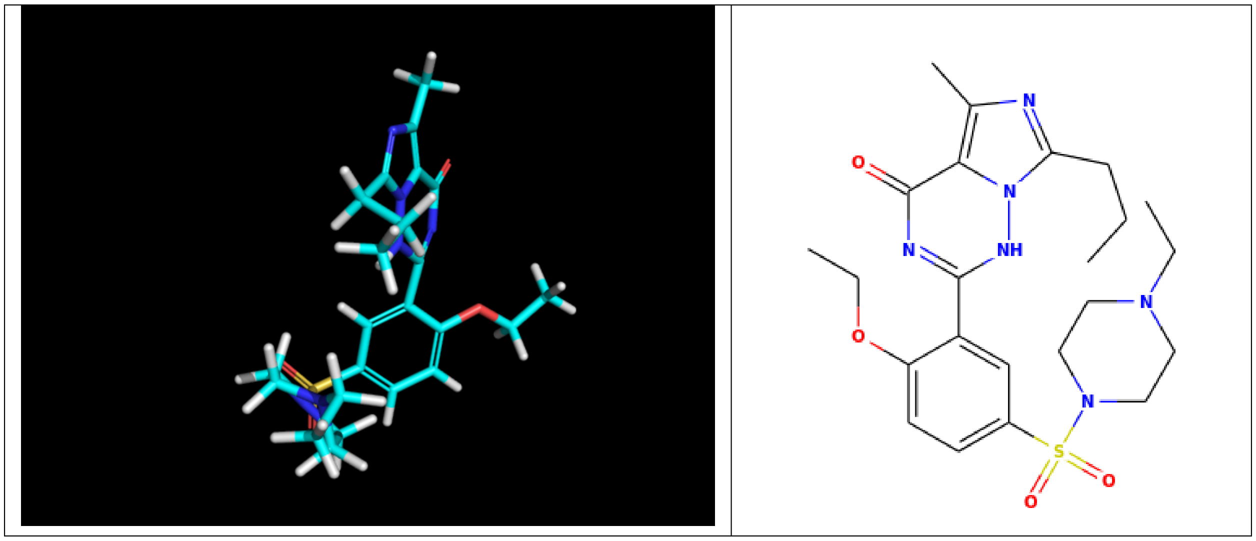
```
>>> template = Chem.MolFromSmiles('c1nccc2n1ccc2')
>>> AllChem.Compute2DCoords(template)
0
>>> AllChem.GenerateDepictionMatching2DStructure(m,template)
```

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.Compute2DCoordsMimicDistmat`, 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.

Working with 3D Molecules

The RDKit can generate conformations for molecules using two different methods. The original method used 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.

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]

More recently, there is an implementation of the method of Riniker and Landrum[18] which uses torsion angle preferences from the Cambridge Structural Database (CSD) to correct the conformers after distance geometry has been used to generate them. With this method, there should be no need to use a minimisation step to clean up the structures.

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)
>>> # use the original distance geometry + minimisation method
>>> AllChem.EmbedMolecule(m2)
0
>>> AllChem.UFFOptimizeMolecule(m2)
```

[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) .

[18] Riniker, S.; Landrum, G. A. "Better Informed Distance Geometry: Using What We Know To Improve Conformation Generation" *J. Chem. Inf. Comp. Sci.* **55**:2562-74 (2015)

```
0
>>> m3=Chem.AddHs(m)
>>> # use the new method
>>> AllChem.EmbedMolecule(m3, AllChem.ETKDG())
0
```

The RDKit also has an implementation of the MMFF94 force field available.[12],[13],[14],[15],[16] Please note that the MMFF atom typing code uses its own aromaticity model, so the aromaticity flags of the molecule will be modified after calling MMFF-related methods.

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

Note the calls to *Chem.AddHs()* in the examples above. By default RDKit molecules do not have H atoms explicitly present in the graph, but they are important for getting realistic geometries, so they generally should be added. They can always be removed afterwards if necessary with a call to *Chem.RemoveHs()*.

With the RDKit, multiple conformers can also be generated using the two different embedding methods. In both cases this is simply a matter of running the distance geometry calculation multiple times from different random start points. The option numConfs allows the user to set the number of conformers that should be generated. Otherwise the procedures are as before. The conformers so generated can be aligned to each other and the RMS values calculated.

```
>>> m = Chem.MolFromSmiles('C1CCC1OC')
>>> m2=Chem.AddHs(m)
>>> # run distance geometry 10 times
>>> cids = AllChem.EmbedMultipleConfs(m2, numConfs=10)
>>> print(len(cids))
10
>>> for cid in cids:
...     _ = AllChem.MMFFOptimizeMolecule(m2, confId=cid)
>>> rmslist = []
>>> AllChem.AlignMolConformers(m2, RMSlist=rmslist)
>>> print(len(rmslist))
9
```

rmslist contains the RMS values between the first conformer and all others. The RMS between two specific conformers (e.g. 1 and 9) can also be calculated. The flag prealigned lets the user specify if the conformers are already aligned (by default, the function aligns them).

```
>>> rms = AllChem.GetConformerRMS(m2, 1, 9, prealigned=True)
```

We can also generate multiple conformers using the new CSD-based method:

```
>>> m = Chem.MolFromSmiles('C1CCC1OC')
>>> m3=Chem.AddHs(m)
```

[12] Halgren, T. A. "Merck molecular force field. I. Basis, form, scope, parameterization, and performance of MMFF94." *J. Comp. Chem.* **17**:490–19 (1996).

[13] Halgren, T. A. "Merck molecular force field. II. MMFF94 van der Waals and electrostatic parameters for intermolecular interactions." *J. Comp. Chem.* **17**:520–52 (1996).

[14] Halgren, T. A. "Merck molecular force field. III. Molecular geometries and vibrational frequencies for MMFF94." *J. Comp. Chem.* **17**:553–86 (1996).

[15] Halgren, T. A. & Nachbar, R. B. "Merck molecular force field. IV. conformational energies and geometries for MMFF94." *J. Comp. Chem.* **17**:587-615 (1996).

[16] Halgren, T. A. "MMFF VI. MMFF94s option for energy minimization studies." *J. Comp. Chem.* **20**:720–9 (1999).

```
>>> # run the new CSD-based method
>>> cids = AllChem.EmbedMultipleConfs(m3, 10, AllChem.ETKDG())
>>> print(len(cids))
10
```

More 3D functionality of the RDKit is described in the Cookbook.

Disclaimer/Warning: Conformation generation is a difficult and subtle task. The original, default, 2D->3D conversion provided with 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. We believe, however, that the newer ETKDG method[#riniker2]_ should be adequate for most purposes.

Preserving Molecules

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

```
>>> m = Chem.MolFromSmiles('c1ccncc1')
>>> import pickle
>>> pkl = pickle.dumps(m)
>>> m2=pickle.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
```

Note that this is smaller than the pickle:

```
>>> len(binStr) < len(pkl)
True
```

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).

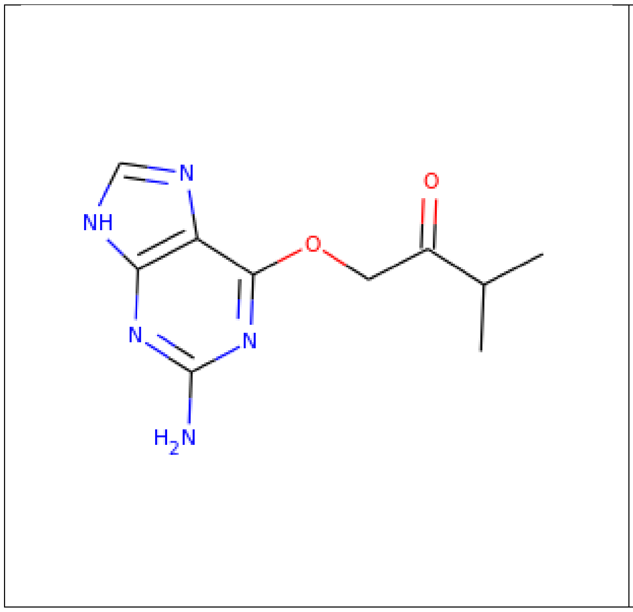
Tip: The performance difference associated with storing molecules in a pickled form on disk instead of constantly reparsing an SD file or SMILES table is difficult to overstate. In a test I just ran on my laptop, loading a set of 699 drug-like molecules from an SD file took 10.8 seconds; loading the 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).

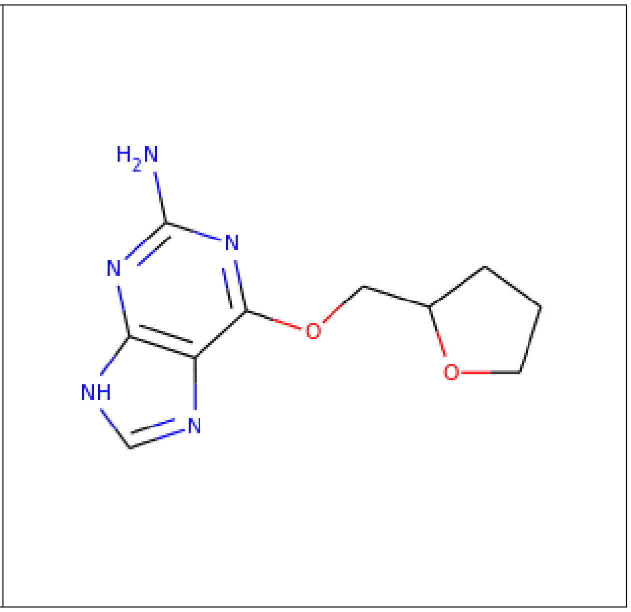
Drawing Molecules

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.o.png')
>>> Draw.MolToFile(ms[1],'images/cdk2_mol2.o.png')
```

Producing these images:





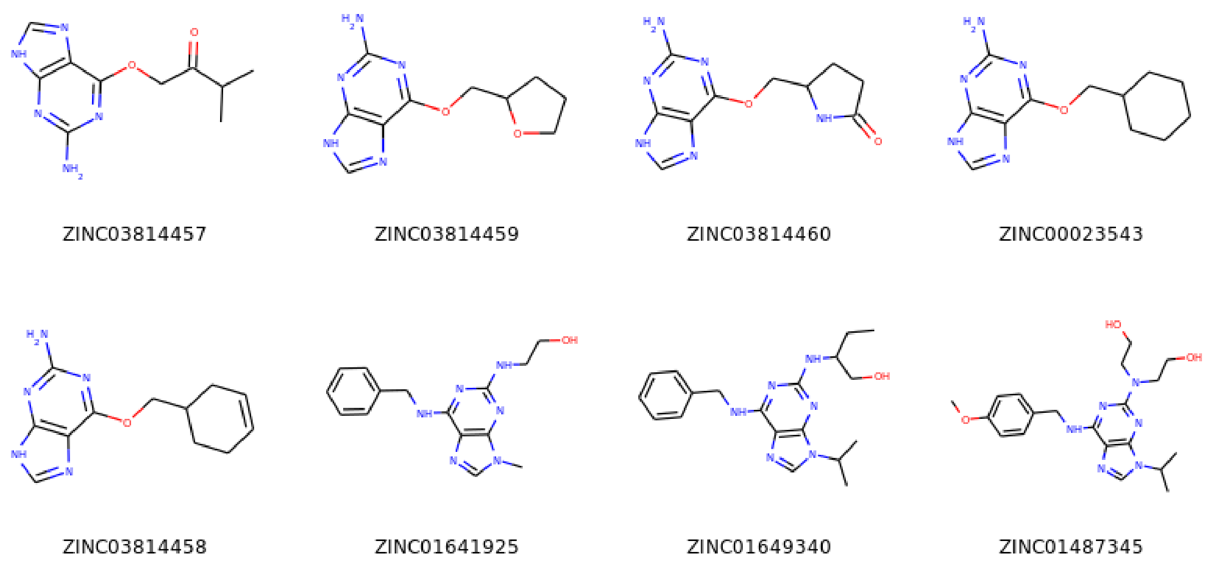
It's also possible to produce an image grid out of a set of molecules:

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

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

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

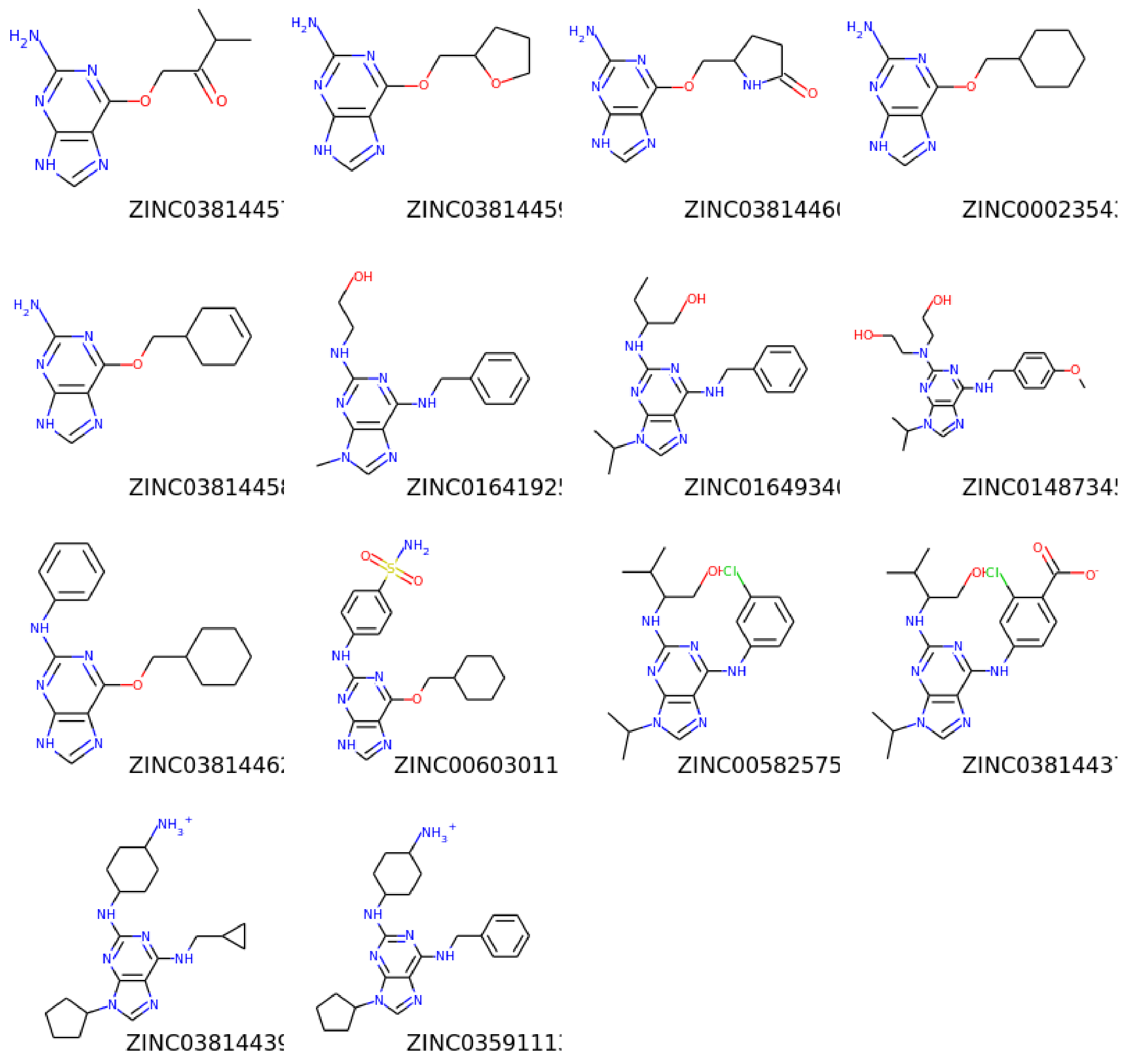
The result looks like this:



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.GenerateDepictionMatching2DStructure(m,p)
>>> img=Draw.MolsToGridImage(subms,molsPerRow=4,subImgSize=(200,200),legends=[x.
↪GetProp("_Name") for x in subms])
>>> img.save('images/cdk2_molgrid.aligned.o.png')
```

The result looks like this:



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
```

Stereochemistry in substructure matches

By default information about stereochemistry is not used in substructure searches:

```
>>> m = Chem.MolFromSmiles('CC[C@H](F)Cl')
>>> m.HasSubstructMatch(Chem.MolFromSmiles('C[C@H](F)Cl'))
True
>>> m.HasSubstructMatch(Chem.MolFromSmiles('C[C@@H](F)Cl'))
True
>>> m.HasSubstructMatch(Chem.MolFromSmiles('CC(F)Cl'))
True
```

But this can be changed via the *useChirality* argument:

```
>>> m.HasSubstructMatch(Chem.MolFromSmiles('C[C@H](F)Cl'),useChirality=True)
True
>>> m.HasSubstructMatch(Chem.MolFromSmiles('C[C@@H](F)Cl'),useChirality=True)
False
```

```
>>> m.HasSubstructMatch(Chem.MolFromSmiles('CC(F)Cl'),useChirality=True)
True
```

Notice that when *useChirality* is set a non-chiral query **does** match a chiral molecule. The same is not true for a chiral query and a non-chiral molecule:

```
>>> m.HasSubstructMatch(Chem.MolFromSmiles('CC(F)Cl'))
True
>>> m2 = Chem.MolFromSmiles('CCC(F)Cl')
>>> m2.HasSubstructMatch(Chem.MolFromSmiles('C[C@H](F)Cl'),useChirality=True)
False
```

Atom Map Indices in SMARTS

It is possible to attach indices to the atoms in the SMARTS pattern. This is most often done in reaction SMARTS (see *Chemical Reactions*), but is more general than that. For example, in the SMARTS patterns for torsion angle analysis published by Guba *et al.* (`DOI: acs.jcim.5b00522`) indices are used to define the four atoms of the torsion of interest. This allows additional atoms to be used to define the environment of the four torsion atoms, as in `[cH0:1][c:2]([cH0])!@[CX3!r:3]=[NX2!r:4]` for an aromatic C=N torsion. We might wonder in passing why they didn't use recursive SMARTS for this, which would have made life easier, but it is what it is. The atom lists from `GetSubstructureMatches` are guaranteed to be in order of the SMARTS, but in this case we'll get five atoms so we need a way of picking out, in the correct order, the four of interest. When the SMARTS is parsed, the relevant atoms are assigned an atom map number property that we can easily extract:

```
>>> qmol = Chem.MolFromSmarts( '[cH0:1][c:2]([cH0])!@[CX3!r:3]=[NX2!r:4]' )
>>> ind_map = {}
>>> for atom in qmol.GetAtoms() :
...     map_num = atom.GetAtomMapNum()
...     if map_num:
...         ind_map[map_num-1] = atom.GetIdx()
>>> ind_map
{0: 0, 1: 1, 2: 3, 3: 4}
>>> map_list = [ind_map[x] for x in sorted(ind_map)]
>>> map_list
[0, 1, 3, 4]
```

Then, when using the query on a molecule you can get the indices of the four matching atoms like this:

```
>>> mol = Chem.MolFromSmiles('Cc1cccc(C)c1C(C)=NC')
>>> for match in mol.GetSubstructMatches( qmol ) :
...     mas = [match[x] for x in map_list]
...     print(mas)
[1, 7, 8, 10]
```

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.

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)
'[*]C(=O)O.[*]CCBr'
```

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'
```

Murcko Decomposition

The RDKit provides standard Murcko-type decomposition[7] 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)
'c1ncc2nc[nH]c2n1'
```

or into a generic framework:

```
>>> fw = MurckoScaffold.MakeScaffoldGeneric(core)
>>> Chem.MolToSmiles(fw)
'C1CCC2CCCC2C1'
```

Maximum Common Substructure

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

```
>>> from rdkit.Chem import rdFMCS
>>> 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]
>>> res=rdFMCS.FindMCS(mols)
>>> res
<rdkit.Chem.rdFMCS.MCSResult object at 0x...>
>>> res.numAtoms
10
>>> res.numBonds
10
>>> res.smartsString
'[#6]1(-[#6]):[#6]:[#6](-[#8]-[#6]):[#6](:[#6]:[#6]:1)-[#8]'
>>> res.canceled
False
```

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 0 and the SMARTS to `''`.

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'))
>>> rdFMCS.FindMCS(mols).smartsString
''
>>> rdFMCS.FindMCS(mols, atomCompare=rdFMCS.AtomCompare.CompareAny).smartsString
```

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

```
'[#7,#8]-[#6]'
>>> rdFMCS.FindMCS(mols, bondCompare=rdFMCS.BondCompare.CompareAny).smartsString
'[#6]-,=[#6]'
```

The options for the atomCompare argument are: CompareAny says that any atom matches any other atom, CompareElements compares by element type, and CompareIsotopes matches based on the isotope label. Isotope labels can be used to implement user-defined atom types. A bondCompare of CompareAny says that any bond matches any other bond, CompareOrderExact says bonds are equivalent if and only if they have the same bond type, and CompareOrder allows single and aromatic bonds to match each other, but requires an exact order match otherwise:

```
>>> mols = (Chem.MolFromSmiles('c1ccccc1'),Chem.MolFromSmiles('C1CCCC=C1'))
>>> rdFMCS.FindMCS(mols,bondCompare=rdFMCS.BondCompare.CompareAny).smartsString
'[#6]1:,-[#6]:,-[#6]:,-[#6]:,-[#6]:,=[#6]:,-1'
>>> rdFMCS.FindMCS(mols,bondCompare=rdFMCS.BondCompare.CompareOrderExact).smartsString
''
>>> rdFMCS.FindMCS(mols,bondCompare=rdFMCS.BondCompare.CompareOrder).smartsString
'[#6](:,-[#6]:,-[#6]:,-[#6]):,-[#6]:,-[#6]'
```

A substructure has both atoms and bonds. By default, the algorithm attempts to maximize the number of bonds found. You can change this by setting the `maximizeBonds` argument to False. 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-]'))
>>> rdFMCS.FindMCS(mols).numAtoms
4
>>> rdFMCS.FindMCS(mols, matchValences=True).numBonds
3
```

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")]
>>> rdFMCS.FindMCS(mols).smartsString
'[#6](-[#6]-[#6])-[#6]-[#6]-[#6]-[#6]'
>>> rdFMCS.FindMCS(mols, ringMatchesRingOnly=True).smartsString
'[#6](-[#6]-[#6])-[#6]'
```

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")]
>>> rdFMCS.FindMCS(mols).smartsString
'[#6]1-[#6]-[#6](-[#6]-1-[#6])-[#6]'
>>> rdFMCS.FindMCS(mols, ringMatchesRingOnly=True).smartsString
'[#6](-[#6]-[#6]-[#6]-[#6])-[#6]'
>>> rdFMCS.FindMCS(mols, completeRingsOnly=True).smartsString
'[#6]1-[#6]-[#6]-[#6]-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 `canceled` property of the MCSResult will be True instead of False.

```
>>> mols = [Chem.MolFromSmiles("Nc1ccccc1"*10), Chem.MolFromSmiles("Nc1ccccccccc1
↪"*10)]
>>> rdFMCS.FindMCS(mols, timeout=1).canceled
True
```

(The MCS after 50 seconds contained 511 atoms.)

Fingerprinting and Molecular Similarity

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

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])
0.6...
>>> DataStructs.FingerprintSimilarity(fps[0],fps[2])
0.4...
>>> 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)
0.75
```

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

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.

Atom Pairs and Topological Torsions

Atom-pair descriptors[3] 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))
```

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:

[3] 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).

```
>>> 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[4] 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.

Morgan Fingerprints (Circular Fingerprints)

This family of fingerprints, better known as circular fingerprints[5], 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:

```
>>> 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)
```

[4] 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).
[5] Rogers, D.; Hahn, M. "Extended-Connectivity Fingerprints." *J. Chem. Inf. and Model.* **50**:742-54 (2010).

```
>>> 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'
```

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.[6] 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
>>> 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:

[6] 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).

```
>>> 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]
```

Generating Similarity Maps Using Fingerprints

Similarity maps are a way to visualize the atomic contributions to the similarity between a molecule and a reference molecule. The methodology is described in Ref.[17] . They are in the `rdkit.Chem.Draw.SimilarityMaps` module :

Start by creating two molecules:

```
>>> from rdkit import Chem
>>> mol = Chem.MolFromSmiles('COc1cccc2cc(C(=O)NCCCCN3CCN(c4cccc5nccnc54)CC3)oc21')
>>> refmol = Chem.MolFromSmiles('CCCN(CCCCN1CCN(c2ccccc2OC)CC1)Cc1ccc2ccccc2c1')
```

The SimilarityMaps module supports three kind of fingerprints: atom pairs, topological torsions and Morgan fingerprints.

```
>>> from rdkit.Chem import Draw
>>> from rdkit.Chem.Draw import SimilarityMaps
>>> fp = SimilarityMaps.GetAPFingerprint(mol, fpType='normal')
>>> fp = SimilarityMaps.GetTTFingerprint(mol, fpType='normal')
>>> fp = SimilarityMaps.GetMorganFingerprint(mol, fpType='bv')
```

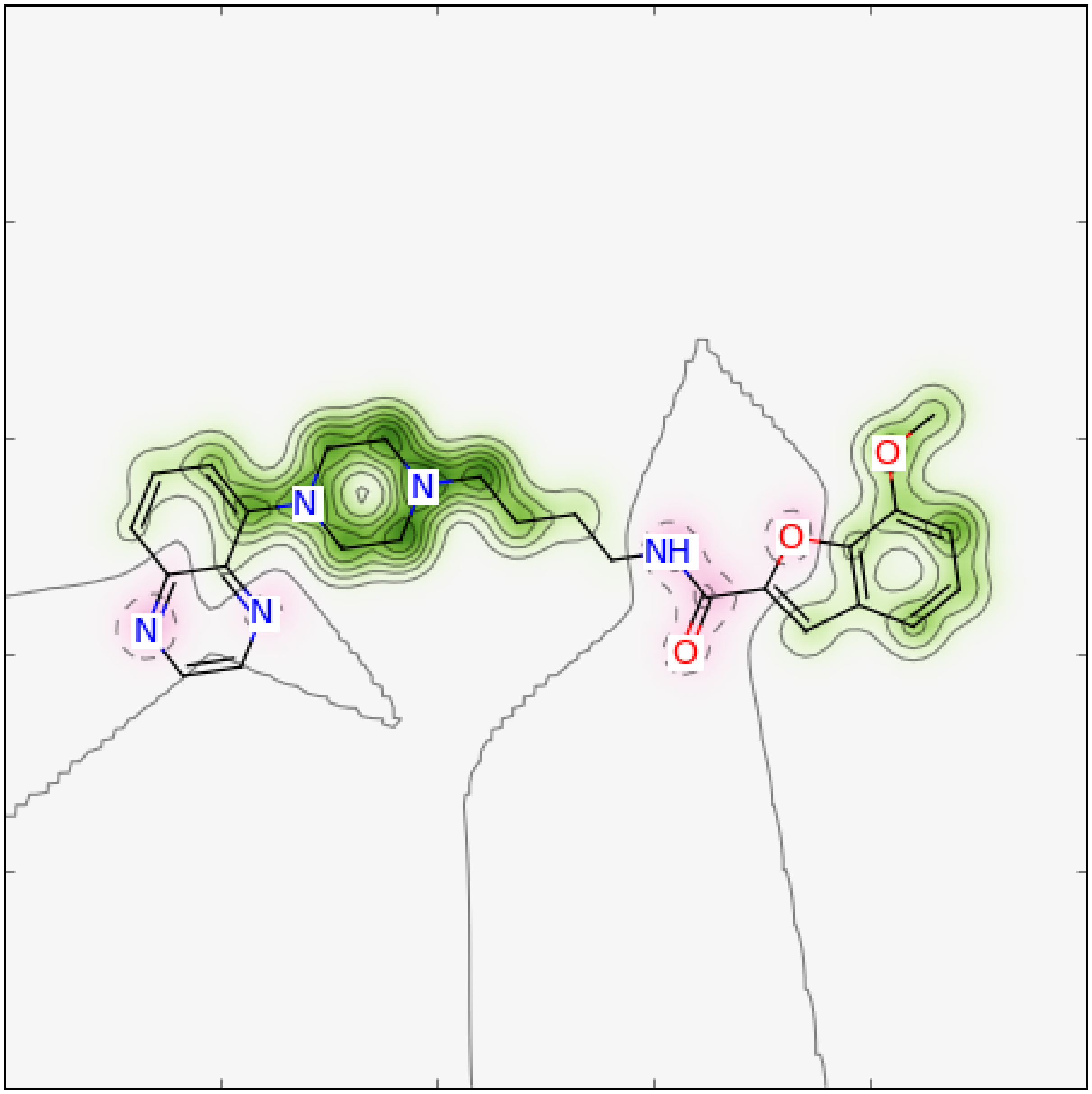
The types of atom pairs and torsions are normal (default), hashed and bit vector (bv). The types of the Morgan fingerprint are bit vector (bv, default) and count vector (count).

The function generating a similarity map for two fingerprints requires the specification of the fingerprint function and optionally the similarity metric. The default for the latter is the Dice similarity. Using all the default arguments of the Morgan fingerprint function, the similarity map can be generated like this:

```
>>> fig, maxweight = SimilarityMaps.GetSimilarityMapForFingerprint(refmol, mol,␣
↪SimilarityMaps.GetMorganFingerprint)
```

Producing this image:

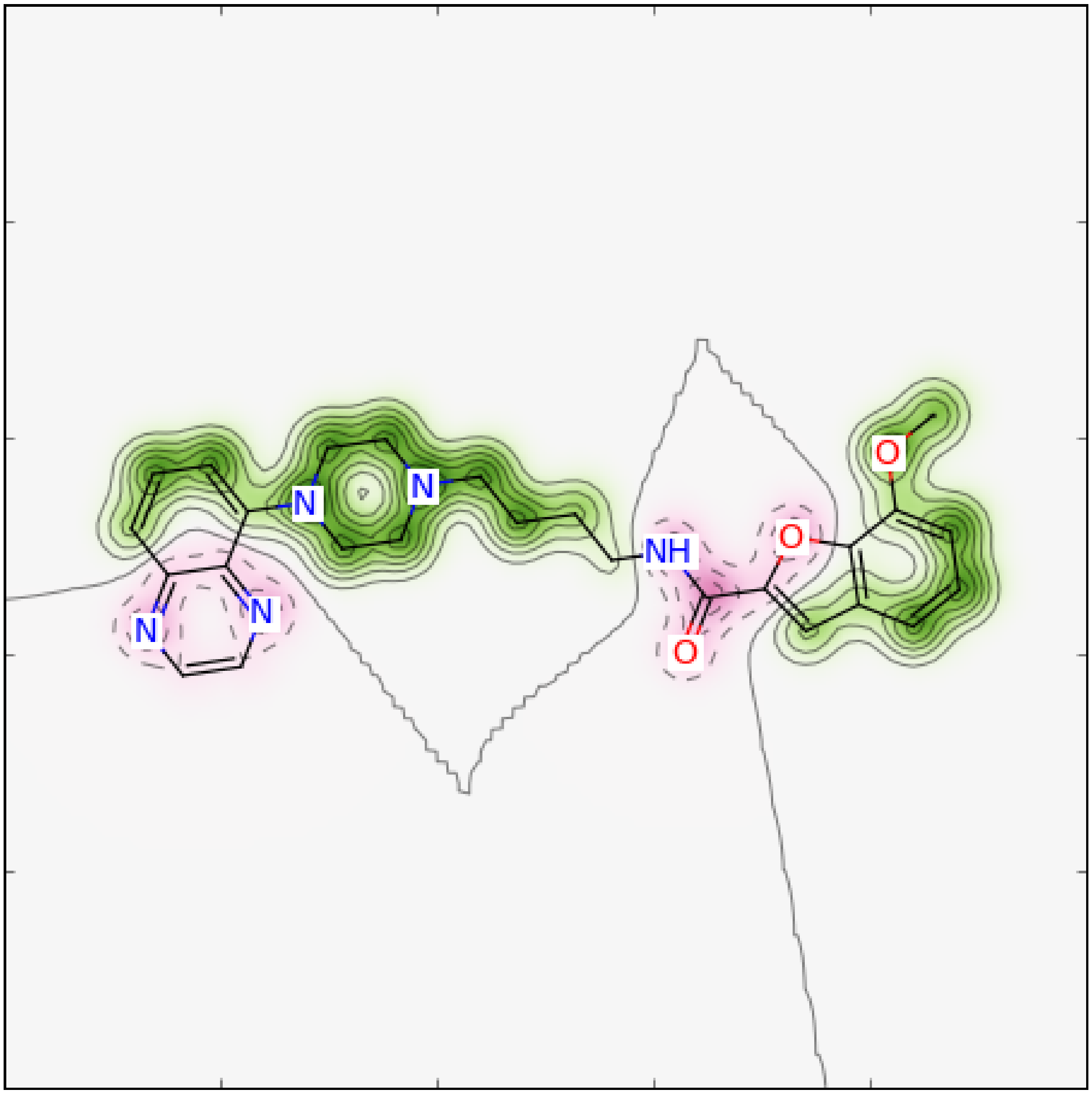
[17] Riniker, S.; Landrum, G. A. "Similarity Maps - A Visualization Strategy for Molecular Fingerprints and Machine-Learning Methods" *J. Cheminf.* **5**:43 (2013).



For a different type of Morgan (e.g. count) and radius = 1 instead of 2, as well as a different similarity metric (e.g. Tanimoto), the call becomes:

```
>>> from rdkit import DataStructs
>>> fig, maxweight = SimilarityMaps.GetSimilarityMapForFingerprint(refmol, mol,␣
↪lambda m,idx: SimilarityMaps.GetMorganFingerprint(m, atomId=idx, radius=1, fpType=
↪'count'), metric=DataStructs.TanimotoSimilarity)
```

Producing this image:



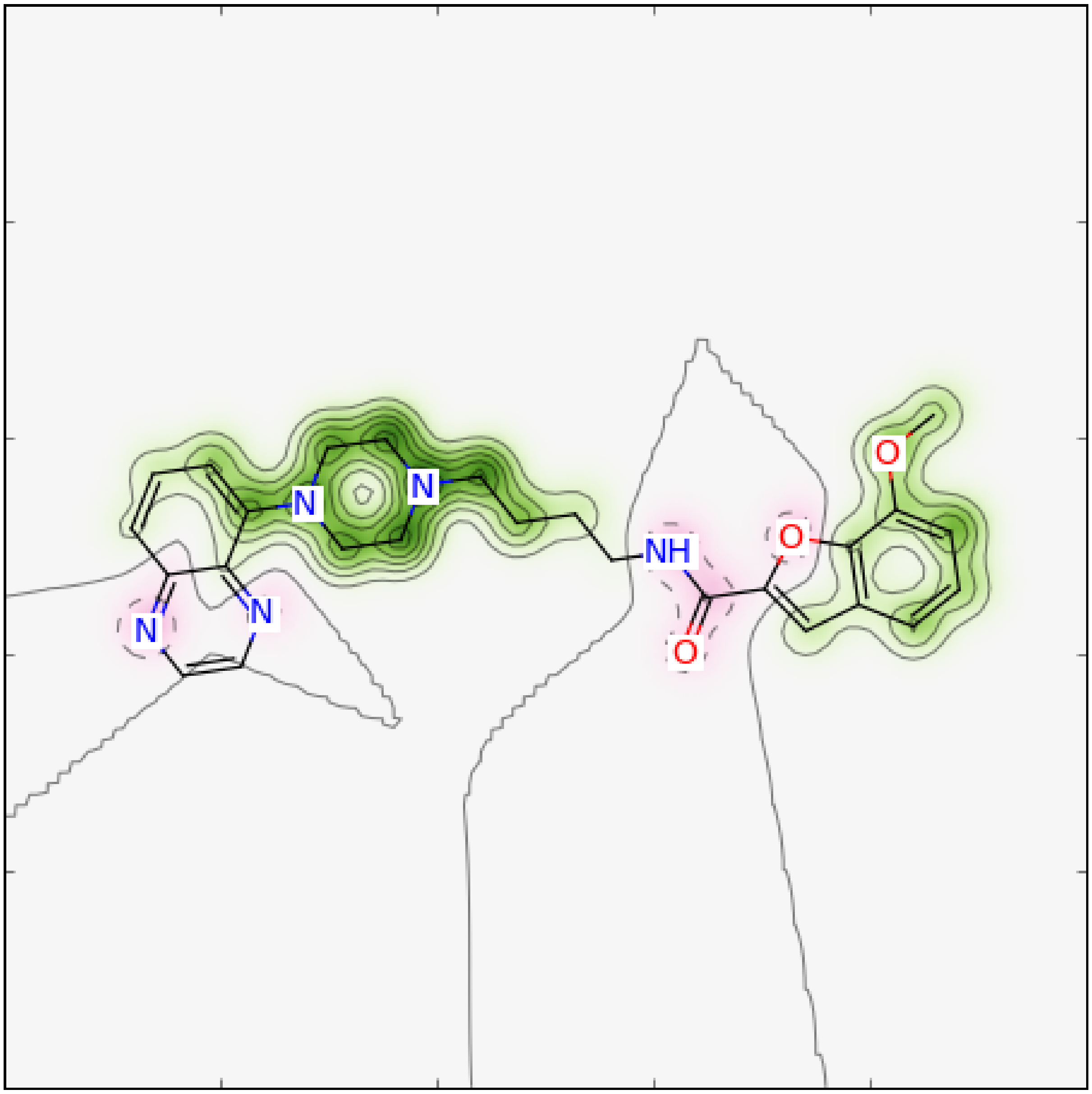
The convenience function GetSimilarityMapForFingerprint involves the normalisation of the atomic weights such that the maximum absolute weight is 1. Therefore, the function outputs the maximum weight that was found when creating the map.

```
>>> print(maxweight)
0.05747...
```

If one does not want the normalisation step, the map can be created like:

```
>>> weights = SimilarityMaps.GetAtomicWeightsForFingerprint(refmol, mol,␣
↪SimilarityMaps.GetMorganFingerprint)
>>> print(["%.2f " % w for w in weights])
['0.05 ', ...
>>> fig = SimilarityMaps.GetSimilarityMapFromWeights(mol, weights)
```

Producing this image:



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...
```

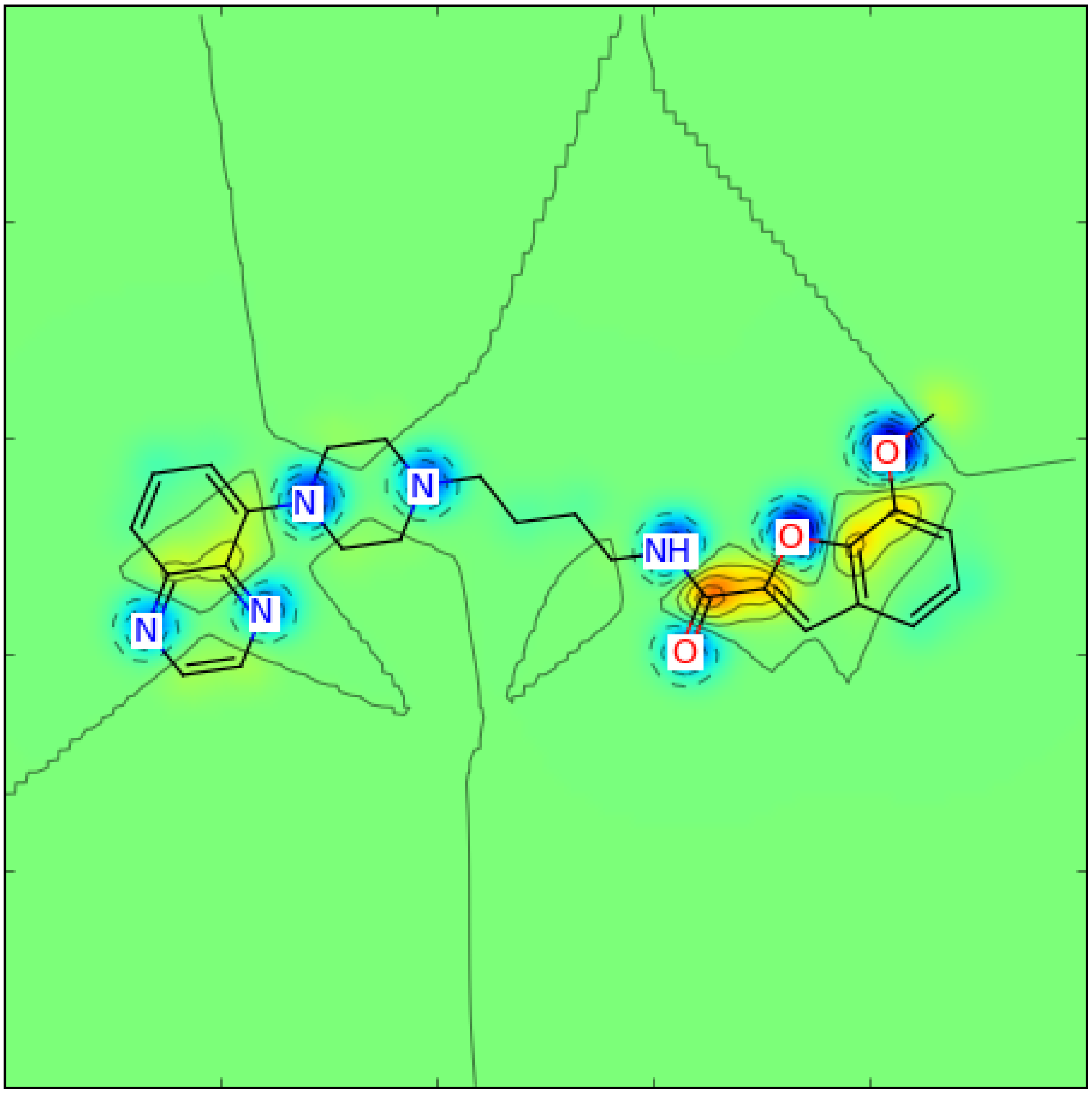
Visualization of Descriptors

Similarity maps can be used to visualize descriptors that can be divided into atomic contributions.

The Gasteiger partial charges can be visualized as (using a different color scheme):

```
>>> from rdkit.Chem.Draw import SimilarityMaps
>>> mol = Chem.MolFromSmiles('COc1cccc2cc(C(=O)NCCCCN3CCN(c4cccc5nccnc54)CC3)oc21')
>>> AllChem.ComputeGasteigerCharges(mol)
>>> contribs = [float(mol.GetAtomWithIdx(i).GetProp('_GasteigerCharge')) for i in␣
↪range(mol.GetNumAtoms())]
>>> fig = SimilarityMaps.GetSimilarityMapFromWeights(mol, contribs, colorMap='jet',␣
↪contourLines=10)
```

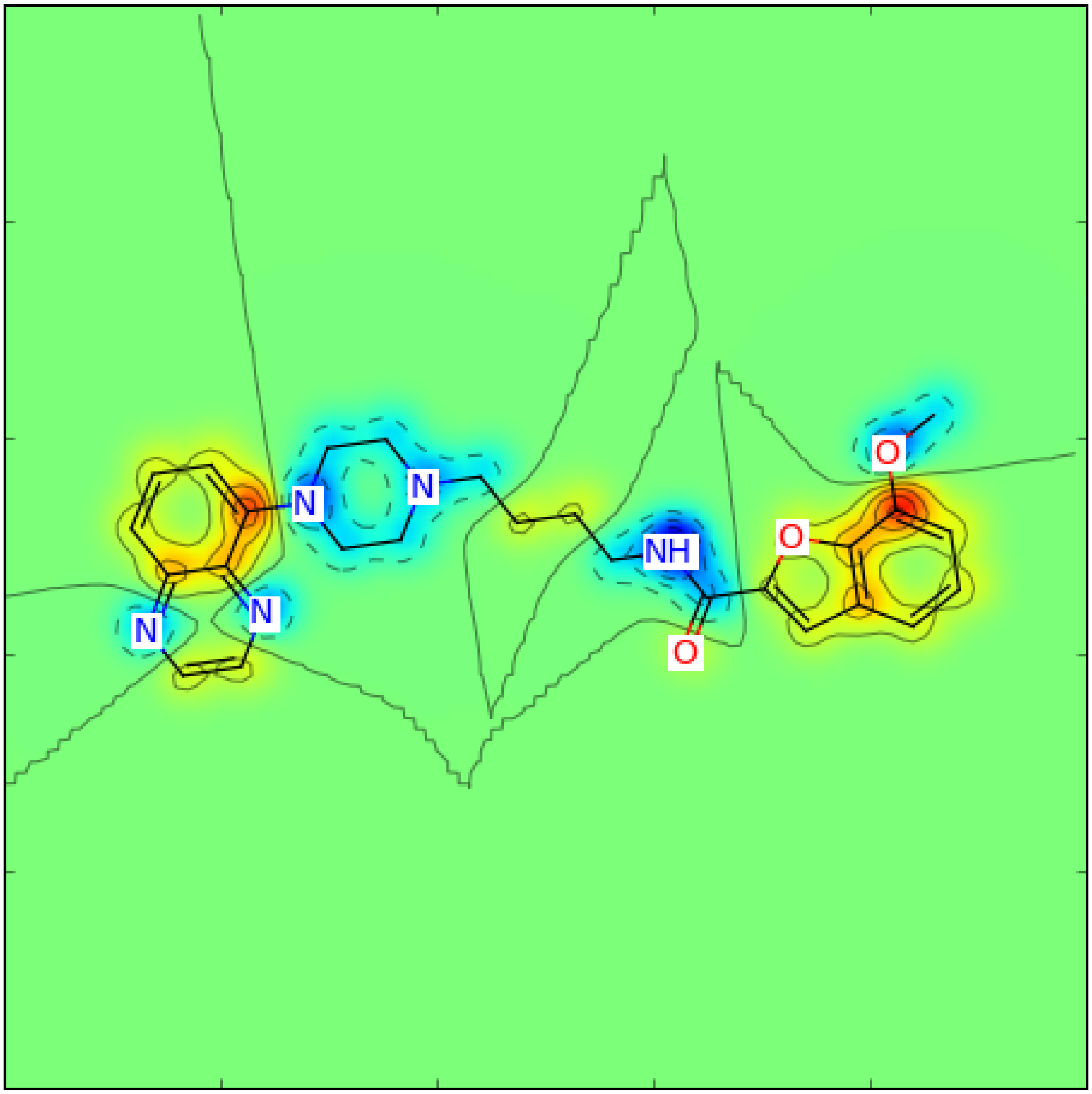
Producing this image:



Or for the Crippen contributions to logP:

```
>>> from rdkit.Chem import rdMolDescriptors
>>> contribs = rdMolDescriptors._CalcCrippenContribs(mol)
>>> fig = SimilarityMaps.GetSimilarityMapFromWeights(mol,[x for x,y in contribs],␣
↪colorMap='jet', contourLines=10)
```

Producing this image:



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[11]:

```
>>> 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
```

[11] 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][C:6]1')
>>> 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]
...
>>> sorted(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]=[C:6]1')
>>> 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'
```

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'
```

Recap Implementation

Associated with the chemical reaction functionality is an implementation of the Recap algorithm.[8] 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:

```
>>> 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()
>>> sorted(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=sorted(ks)
>>> 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'
```

BRICS Implementation

The RDKit also provides an implementation of the BRICS algorithm.[9] 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]
>>> sorted(BRICS.BRICSDecompose(m1))
['[14*]c1nc(N)nc2[nH]cnc12', '[3*]O[3*]', '[4*]CC(=O)C(C)C']
>>> m2 = cdk2mols[20]
```

[8] 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).

[9] 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).

```
>>> sorted(BRICS.BRICSDecompose(m2))
['[1*]C(=O)NN(C)C', '[14*]c1[nH]nc2c1C(=O)c1c([16*])cccc1-2', '[16*]c1ccc([16*])cc1',
↪'[3*]OC', '[5*]N[5*]']
```

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)
...    allfrags.update(pieces)
>>> len(allfrags)
90
>>> sorted(allfrags)[:5]
['NS(=O)(=O)c1ccc(N/N=C2\\C(=O)Nc3ccc(Br)cc32)cc1', '[1*]C(=O)C(C)C', '[1*]C(=O)NN(C)C
↪', '[1*]C(=O)NN1CC[NH+](C)CC1', '[1*]C(C)=O']
```

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 sorted(allfrags)]
>>> ms = BRICS.BRICSBuild(fragms)
```

The result is a generator object:

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

That returns molecules on request:

```
>>> prods = [next(ms) for x in range(10)]
>>> prods[0]
<rdkit.Chem.rdchem.Mol object at 0x...>
```

The molecules have not been sanitized, so it's a good idea to at least update the valences before continuing:

```
>>> for prod in prods:
...     prod.UpdatePropertyCache(strict=False)
...
>>> Chem.MolToSmiles(prods[0],True)
'COCCO'
>>> Chem.MolToSmiles(prods[1],True)
'O=C1Nc2ccc3ncsc3c2/C1=C/NCCO'
>>> Chem.MolToSmiles(prods[2],True)
'O=C1Nc2ccccc2/C1=C/NCCO'
```

Other fragmentation approaches

In addition to the methods described above, the RDKit provide a very flexible generic function for fragmenting molecules along user-specified bonds.

Here's a quick demonstration of using that to break all bonds between atoms in rings and atoms not in rings. We start by finding all the atom pairs:

```
>>> m = Chem.MolFromSmiles('CC1CC(O)C1CCC1CC1')
>>> bis = m.GetSubstructMatches(Chem.MolFromSmarts('[!R][R]'))
>>> bis
((0, 1), (4, 3), (6, 5), (7, 8))
```

then we get the corresponding bond indices:

```
>>> bs = [m.GetBondBetweenAtoms(x,y).GetIdx() for x,y in bis]
>>> bs
[0, 3, 5, 7]
```

then we use those bond indices as input to the fragmentation function:

```
>>> nm = Chem.FragmentOnBonds(m,bs)
```

the output is a molecule that has dummy atoms marking the places where bonds were broken:

```
>>> Chem.MolToSmiles(nm,True)
'[*]C1CC([4*])C1[6*].[1*]C.[3*]O.[5*]CC[8*].[7*]C1CC1'
```

By default the attachment points are labelled (using isotopes) with the index of the atom that was removed. We can also provide our own set of atom labels in the form of pairs of unsigned integers. The first value in each pair is used as the label for the dummy that replaces the bond's begin atom, the second value in each pair is for the dummy that replaces the bond's end atom. Here's an example, repeating the analysis above and marking the positions where the non-ring atoms were with the label 10 and marking the positions where the ring atoms were with label 1:

```
>>> bis = m.GetSubstructMatches(Chem.MolFromSmarts('[!R][R]'))
>>> bs = []
>>> labels=[]
>>> for bi in bis:
...    b = m.GetBondBetweenAtoms(bi[0],bi[1])
...    if b.GetBeginAtomIdx()==bi[0]:
...        labels.append((10,1))
...    else:
...        labels.append((1,10))
...    bs.append(b.GetIdx())
>>> nm = Chem.FragmentOnBonds(m,bs,dummyLabels=labels)
>>> Chem.MolToSmiles(nm,True)
'[1*]C.[1*]CC[1*].[1*]O.[10*]C1CC([10*])C1[10*].[10*]C1CC1'
```

Chemical Features and Pharmacophores

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())
[2.07..., -2.335..., 0.0]
```

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.[10] 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|'
```

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

```
>>> Gobbi_Pharm2D.factory.GetBitDescription(30184)
'HA HD HD |0 3 0|3 0 3|0 3 0|'
```

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

```
'C<-O>C'
>>> fcat.GetEntryDescription(1)
'CN<-cPropyl>'
>>> list(fcat.GetEntryDownIds(0))
[3, 4]
>>> fcat.GetEntryDescription(3)
'C<-O>CC'
>>> 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.

Non-Chemical Functionality

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.

Discrete value vectors

3D grids

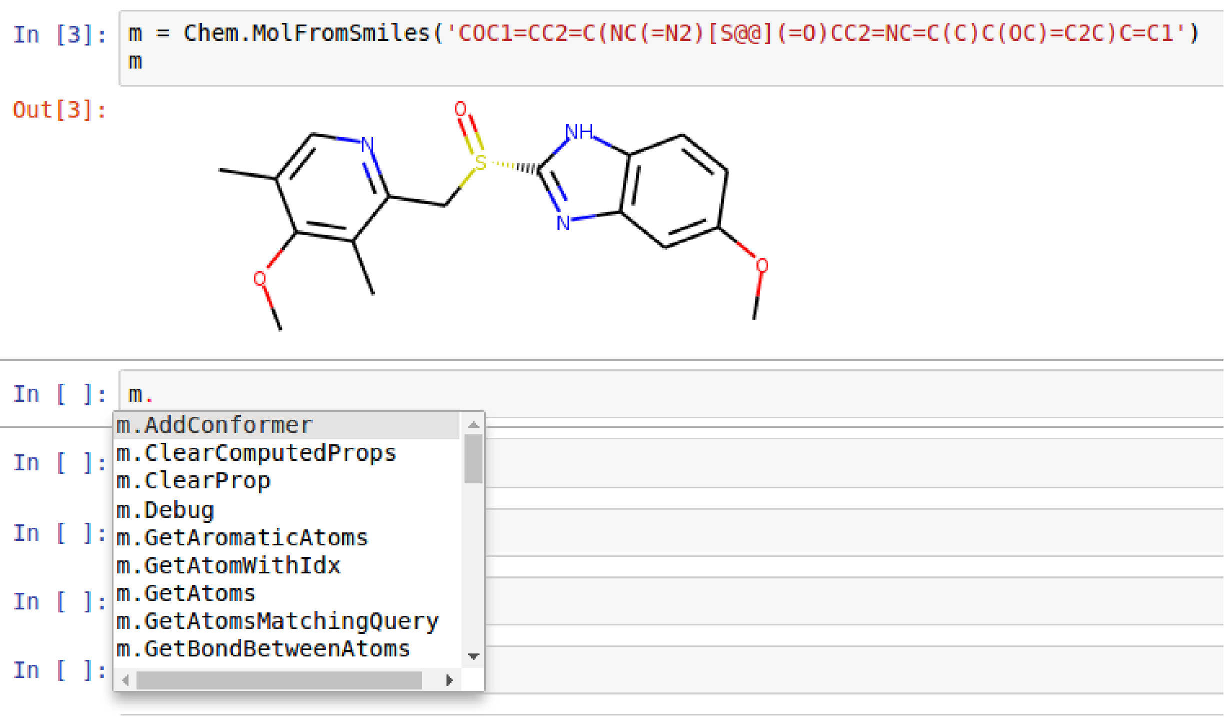
Points

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 the Jupyter notebook:



```
In [3]: m = Chem.MolFromSmiles('COC1=CC2=C(NC(=N2)[S@@](=O)CC2=NC=C(C)C(OC)=C2C)C=C1')
        m
Out[3]:
```

```
In [ ]: m.
        m.AddConformer
        m.ClearComputedProps
        m.ClearProp
        m.Debug
        m.GetAromaticAtoms
        m.GetAtomWithIdx
        m.GetAtoms
        m.GetAtomsMatchingQuery
        m.GetBondBetweenAtoms
In [ ]:
In [ ]:
In [ ]:
In [ ]:
```

Advanced Topics/Warnings

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.RWMol` class:

```
>>> m = Chem.MolFromSmiles('CC(=O)C=CC=C')
>>> mw = Chem.RWMol(m)
>>> mw.ReplaceAtom(4,Chem.Atom(7))
>>> mw.AddAtom(Chem.Atom(6))
7
>>> mw.AddAtom(Chem.Atom(6))
8
>>> mw.AddBond(6,7,Chem.BondType.SINGLE)
7
>>> mw.AddBond(7,8,Chem.BondType.DOUBLE)
8
>>> mw.AddBond(8,3,Chem.BondType.SINGLE)
9
>>> mw.RemoveAtom(0)
>>> mw.GetNumAtoms()
8
```

The RWMol can be used just like an ROMol:

```
>>> Chem.MolToSmiles(mw)
'O=CC1C=CC=CN=1'
>>> Chem.SanitizeMol(mw)
rdkit.Chem.rdmolops.SanitizeFlags.SANITIZE_NONE
>>> Chem.MolToSmiles(mw)
'O=Cc1ccccn1'
```

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

Miscellaneous Tips and Hints

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')
```

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.

List of Available Descriptors

Descriptor/Descriptor Family	Notes	Language
Gasteiger/Marsili Partial Charges	*Tetrahedron* **36**:3219-28 (1980)	C++
BalabanJ	*Chem. Phys. Lett.* **89**:399-404 (1982)	Python
BertzCT	*J. Am. Chem. Soc.* **103**:3599-601 (1981)	Python
Ipc	*J. Chem. Phys.* **67**:4517-33 (1977)	Python
HallKierAlpha	*Rev. Comput. Chem.* **2**:367-422 (1991)	C++
Kappa1 - Kappa3	*Rev. Comput. Chem.* **2**:367-422 (1991)	C++
Chi0, Chi1	*Rev. Comput. Chem.* **2**:367-422 (1991)	Python
Chi0n - Chi4n	*Rev. Comput. Chem.* **2**:367-422 (1991)	C++
Chi0v - Chi4v	*Rev. Comput. Chem.* **2**:367-422 (1991)	C++

Continued on next page

Table 3.1 – continued from previous page

MolLogP	Wildman and Crippen *JCICS* **39**:868-73 (1999)	C++
MolMR	Wildman and Crippen *JCICS* **39**:868-73 (1999)	C++
MolWt		C++
ExactMolWt		C++
HeavyAtomCount		C++
HeavyAtomMolWt		C++
NHOHCount		C++
NOCount		C++
NumHAcceptors		C++
NumHDonors		C++
NumHeteroatoms		C++
NumRotatableBonds		C++
NumValenceElectrons		C++
NumAmideBonds		C++
Num{Aromatic,Saturated,Aliphatic}Rings		C++
Num{Aromatic,Saturated,Aliphatic}{Hetero,Carbo}cycles		C++
RingCount		C++
FractionCSP3		C++
NumSpiroAtoms	Number of spiro atoms (atoms shared between rings that share exactly one atom)	C++
NumBridgeheadAtoms	Number of bridgehead atoms (atoms shared between rings that share at least two bonds)	C++
TPSA	*J. Med. Chem.* **43**:3714-7, (2000)	C++
LabuteASA	*J. Mol. Graph. Mod.* **18**:464-77 (2000)	C++
PEOE_VSA1 - PEOE_VSA14	MOE-type descriptors using partial charges and surface area contributions http://www.chemcomp.com/journal/vsadesc.htm	C++
SMR_VSA1 - SMR_VSA10	MOE-type descriptors using MR contributions and surface area contributions http://www.chemcomp.com/journal/vsadesc.htm	C++
SlogP_VSA1 - SlogP_VSA12	MOE-type descriptors using LogP contributions and surface area contributions http://www.chemcomp.com/journal/vsadesc.htm	C++
EState_VSA1 - EState_VSA11	MOE-type descriptors using EState indices and surface area contributions (developed at RD, not described in the CCG paper)	Python
VSA_EState1 - VSA_EState10	MOE-type descriptors using EState indices and surface area contributions (developed at RD, not described in the CCG paper)	Python

Continued on next page

Table 3.1 – continued from previous page

MQNs	Nguyen et al. *ChemMedChem* **4**:1803-5 (2009)	C++
Topliss fragments	implemented using a set of SMARTS definitions in $(RD-BASE)/Data/FragmentDescriptors.csv	Python

List of Available Fingerprints

Fingerprint Type	Notes	Language
RDKit	a Daylight-like fingerprint based on hashing molecular subgraphs	C++
Atom Pairs	*JCICS* **25**:64-73 (1985)	C++
Topological Torsions	*JCICS* **27**:82-5 (1987)	C++
MACCS keys	Using the 166 public keys implemented as SMARTS	C++
Morgan/Circular	Fingerprints based on the Morgan algorithm, similar to the ECFP/FCFP fingerprints *JCIM* **50**:742-54 (2010).	C++
2D Pharmacophore	Uses topological distances between pharmacophoric points.	C++
Pattern	a topological fingerprint optimized for substructure screening	C++
Extended Reduced Graphs	Derived from the ErG fingerprint published by Stiefl et al. in *JCIM* **46**:208–20 (2006). NOTE: these functions return an array of floats, not the usual fingerprint types	C++

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])]),n&H0&+0,$([o,s;+0;!$([o,s]:n);!$([o,s]:c:n)])]`
Aromatic	`[a]`
Halogen	`[F,Cl,Br,I]`
Basic	`[#7;+,$([N;H2&+0][$([C,a]);!$([C,a](=O))]),$([N;H1&+0]([$([C,a]);!$([C,a](=O))])[$([C,a]);!$([C,a](=O))]),$([N;H0&+0]([C;!$(C(=O))])([C;!$(C(=O))])[C;!$(C(=O))])]`
Acidic	`[$([C,S](=[O,S,P])-[O;H1,-1])]`

License

CHAPTER 4

The RDKit Book

Misc Cheminformatics Topics

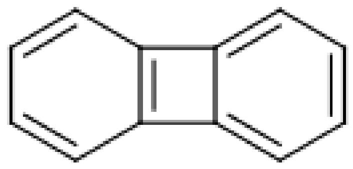
Aromaticity

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
```

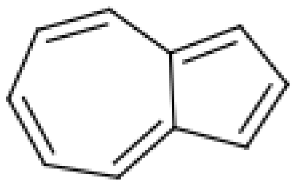
A ring, or fused ring system, is considered to be aromatic if it obeys the 4N+2 rule. Contributions to the electron count are determined by atom type and environment. Some examples:

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

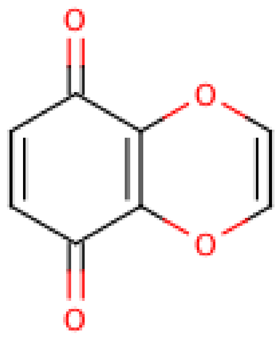
Notation a: any aromatic atom; A: any atom, include H; *: a dummy atom

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:



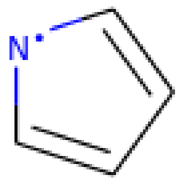
An extreme example, demonstrating both fused rings and the influence of exocyclic double bonds:



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

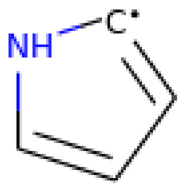
```
>>> m.GetBondBetweenAtoms(6,7).GetIsAromatic()
False
```

A special case, heteroatoms with radicals are not considered candidates for aromaticity:



```
>>> m = Chem.MolFromSmiles('C1=C[N]C=C1')
>>> m.GetAtomWithIdx(0).GetIsAromatic()
False
>>> m.GetAtomWithIdx(2).GetIsAromatic()
False
>>> m.GetAtomWithIdx(2).GetNumRadicalElectrons()
1
```

Carbons with radicals, however, are still considered:



```
>>> m = Chem.MolFromSmiles('C1=[C]NC=C1')
>>> m.GetAtomWithIdx(0).GetIsAromatic()
True
>>> m.GetAtomWithIdx(1).GetIsAromatic()
True
```

```
>>> m.GetAtomWithIdx(1).GetNumRadicalElectrons()
1
```

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.

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.

Chemical Reaction Handling

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
                  ``('' 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']
```

[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

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

Intramolecular reactions can be expressed flexibly by including reactants in parentheses. This is demonstrated in this ring-closing metathesis example[4]:

```
>>> rxn = AllChem.ReactionFromSmarts("([C:1]=[C;H2].[C:2]=[C;H2])>>[*:1]=[*:2]")
>>> m1 = Chem.MolFromSmiles('C=CCOCC=C')
>>> ps = rxn.RunReactants((m1,))
>>> Chem.MolToSmiles(ps[0][0])
'C1=CCOC1'
```

Chirality

This section describes how chirality information in the reaction defition is handled. A consistent example, esterification of secondary alcohols, is used throughout[5].

If no chiral information is present in the reaction definition, the stereochemistry of the reactants is preserved:

```
>>> alcohol1 = Chem.MolFromSmiles('CC(CCN)O')
>>> alcohol2 = Chem.MolFromSmiles('C[C@H](CCN)O')
>>> alcohol3 = Chem.MolFromSmiles('C[C@@H](CCN)O')
>>> acid = Chem.MolFromSmiles('CC(=O)O')
>>> rxn = AllChem.ReactionFromSmarts('[CH1:1][OH:2].[OH][C:3]=[O:4]>>
↪[C:1][O:2][C:3]=[O:4]')
>>> ps=rxn.RunReactants((alcohol1,acid))
>>> Chem.MolToSmiles(ps[0][0],True)
'CC(=O)OC(C)CCN'
>>> ps=rxn.RunReactants((alcohol2,acid))
>>> Chem.MolToSmiles(ps[0][0],True)
'CC(=O)O[C@H](C)CCN'
>>> ps=rxn.RunReactants((alcohol3,acid))
>>> Chem.MolToSmiles(ps[0][0],True)
'CC(=O)O[C@@H](C)CCN'
```

You get the same result (retention of stereochemistry) if a mapped atom has the same chirality in both reactants and products:

```
>>> rxn = AllChem.ReactionFromSmarts('[C@H1:1][OH:2].[OH][C:3]=[O:4]>>
↪[C@:1][O:2][C:3]=[O:4]')
>>> ps=rxn.RunReactants((alcohol1,acid))
```

[4] Thanks to James Davidson for this example.

[5] Thanks to JP Ebejer and Paul Finn for this example.

```
>>> Chem.MolToSmiles(ps[0][0],True)
'CC(=O)OC(C)CCN'
>>> ps=rxn.RunReactants((alcohol2,acid))
>>> Chem.MolToSmiles(ps[0][0],True)
'CC(=O)O[C@H](C)CCN'
>>> ps=rxn.RunReactants((alcohol3,acid))
>>> Chem.MolToSmiles(ps[0][0],True)
'CC(=O)O[C@@H](C)CCN'
```

A mapped atom with different chirality in reactants and products leads to inversion of stereochemistry:

```
>>> rxn = AllChem.ReactionFromSmarts('[C@H1:1][OH:2].[OH][C:3]=[O:4]>>
↪[C@@:1][O:2][C:3]=[O:4]')
>>> ps=rxn.RunReactants((alcohol1,acid))
>>> Chem.MolToSmiles(ps[0][0],True)
'CC(=O)OC(C)CCN'
>>> ps=rxn.RunReactants((alcohol2,acid))
>>> Chem.MolToSmiles(ps[0][0],True)
'CC(=O)O[C@@H](C)CCN'
>>> ps=rxn.RunReactants((alcohol3,acid))
>>> Chem.MolToSmiles(ps[0][0],True)
'CC(=O)O[C@H](C)CCN'
```

If a mapped atom has chirality specified in the reactants, but not in the products, the reaction destroys chirality at that center:

```
>>> rxn = AllChem.ReactionFromSmarts('[C@H1:1][OH:2].[OH][C:3]=[O:4]>>
↪[C:1][O:2][C:3]=[O:4]')
>>> ps=rxn.RunReactants((alcohol1,acid))
>>> Chem.MolToSmiles(ps[0][0],True)
'CC(=O)OC(C)CCN'
>>> ps=rxn.RunReactants((alcohol2,acid))
>>> Chem.MolToSmiles(ps[0][0],True)
'CC(=O)OC(C)CCN'
>>> ps=rxn.RunReactants((alcohol3,acid))
>>> Chem.MolToSmiles(ps[0][0],True)
'CC(=O)OC(C)CCN'
```

And, finally, if chirality is specified in the products, but not the reactants, the reaction creates a stereocenter with the specified chirality:

```
>>> rxn = AllChem.ReactionFromSmarts('[CH1:1][OH:2].[OH][C:3]=[O:4]>>
↪[C@:1][O:2][C:3]=[O:4]')
>>> ps=rxn.RunReactants((alcohol1,acid))
>>> Chem.MolToSmiles(ps[0][0],True)
'CC(=O)O[C@H](C)CCN'
>>> ps=rxn.RunReactants((alcohol2,acid))
>>> Chem.MolToSmiles(ps[0][0],True)
'CC(=O)O[C@H](C)CCN'
>>> ps=rxn.RunReactants((alcohol3,acid))
>>> Chem.MolToSmiles(ps[0][0],True)
'CC(=O)O[C@H](C)CCN'
```

Note that this doesn’t make sense without including a bit more context around the stereocenter in the reaction definition:

```
>>> rxn = AllChem.ReactionFromSmarts('[CH3:5][CH1:1]([C:6])[OH:2].[OH][C:3]=[O:4]>>
↪[C:5][C@:1]([C:6])[O:2][C:3]=[O:4]')
```

```
>>> ps=rxn.RunReactants((alcohol1,acid))
>>> Chem.MolToSmiles(ps[0][0],True)
'CC(=O)O[C@H](C)CCN'
>>> ps=rxn.RunReactants((alcohol2,acid))
>>> Chem.MolToSmiles(ps[0][0],True)
'CC(=O)O[C@H](C)CCN'
>>> ps=rxn.RunReactants((alcohol3,acid))
>>> Chem.MolToSmiles(ps[0][0],True)
'CC(=O)O[C@H](C)CCN'
```

Note that the chirality specification is not being used as part of the query: a molecule with no chirality specified can match a reactant with specified chirality.

In general, the reaction machinery tries to preserve as much stereochemistry information as possible. This works when a single new bond is formed to a chiral center:

```
>>> rxn = AllChem.ReactionFromSmarts('[C:1][C:2]-O>>[C:1][C:2]-S')
>>> alcohol2 = Chem.MolFromSmiles('C[C@@H](O)CCN')
>>> ps=rxn.RunReactants((alcohol2,))
>>> Chem.MolToSmiles(ps[0][0],True)
'C[C@@H](S)CCN'
```

But it fails if two or more bonds are formed:

```
>>> rxn = AllChem.ReactionFromSmarts('[C:1][C:2](-O)-F>>[C:1][C:2](-S)-Cl')
>>> alcohol = Chem.MolFromSmiles('C[C@@H](O)F')
>>> ps=rxn.RunReactants((alcohol,))
>>> Chem.MolToSmiles(ps[0][0],True)
'CC(S)Cl'
```

In this case, there's just not sufficient information present to allow the information to be preserved. You can help by providing mapping information:

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]]
['c1ccccc-1']
```

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']
```

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.

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.

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

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.

Representation of Pharmacophore Fingerprints

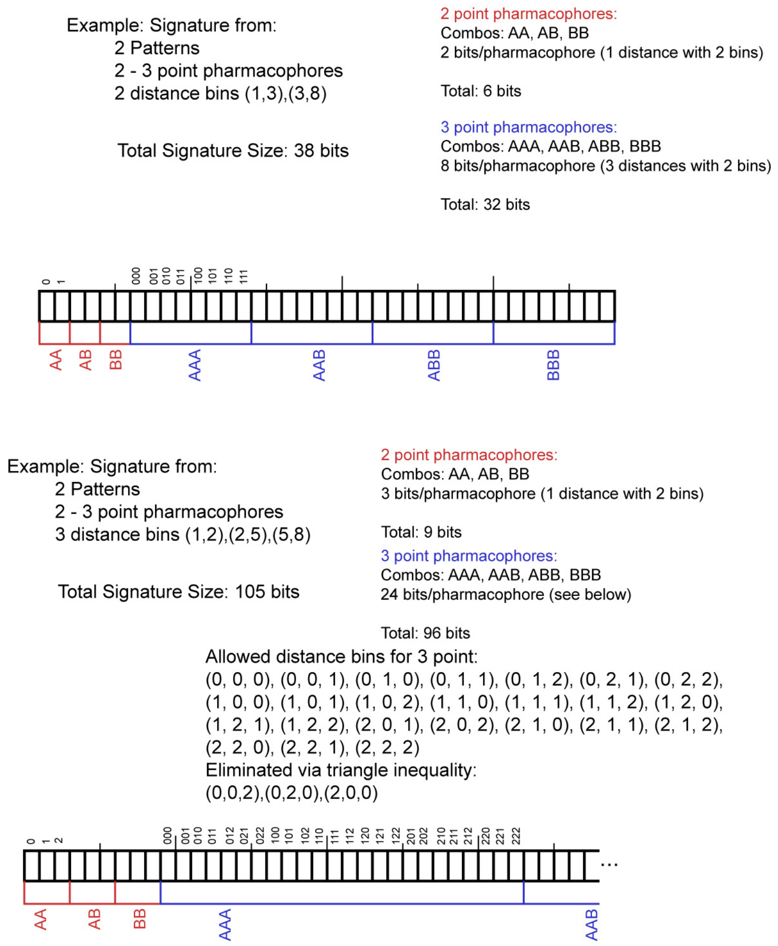
In the RDKit scheme the bit ids in pharmacophore fingerprints are not hashed: each bit corresponds to a particular combination of features and distances. A given bit id can be converted back to the corresponding feature types and distances to allow interpretation. An illustration for 2D pharmacophores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

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:

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]	No
OCO	[CH2]	No
OCO	[CH3]	No
OCO	O[CH3]	Yes
O[CH2]O	C	Yes
O[CH2]O	[CH2]	No



Example: Signature from:
2 Patterns
2 - 3 point pharmacophores
2 distance bins (1,3),(3,8)

Total Signature Size: 38 bits

2 point pharmacophores:
Combos: AA, AB, BB
2 bits/pharmacophore (1 distance with 2 bins)

Total: 6 bits

3 point pharmacophores:
Combos: AAA, AAB, ABB, BBB
8 bits/pharmacophore (3 distances with 2 bins)

Total: 32 bits

Example: Signature from:
2 Patterns
2 - 3 point pharmacophores
3 distance bins (1,2),(2,5),(5,8)

Total Signature Size: 105 bits

2 point pharmacophores:
Combos: AA, AB, BB
3 bits/pharmacophore (1 distance with 2 bins)

Total: 9 bits

3 point pharmacophores:
Combos: AAA, AAB, ABB, BBB
24 bits/pharmacophore (see below)

Total: 96 bits

Allowed distance bins for 3 point:
(0, 0, 0), (0, 0, 1), (0, 1, 0), (0, 1, 1), (0, 1, 2), (0, 2, 1), (0, 2, 2),
(1, 0, 0), (1, 0, 1), (1, 0, 2), (1, 1, 0), (1, 1, 1), (1, 1, 2), (1, 2, 0),
(1, 2, 1), (1, 2, 2), (2, 0, 1), (2, 0, 2), (2, 1, 0), (2, 1, 1), (2, 1, 2),
(2, 2, 0), (2, 2, 1), (2, 2, 2)
Eliminated via triangle inequality:
(0,0,2),(0,2,0),(2,0,0)

Fig. 4.1: Figure 1: Bit numbering in pharmacophore fingerprints

Demonstrated here:

```
>>> Chem.MolFromSmiles('CCO').HasSubstructMatch(Chem.MolFromSmiles('CCO'))
True
>>> Chem.MolFromSmiles('CC[O-]').HasSubstructMatch(Chem.MolFromSmiles('CCO'))
True
>>> Chem.MolFromSmiles('CCO').HasSubstructMatch(Chem.MolFromSmiles('CC[O-]'))
False
>>> Chem.MolFromSmiles('CC[O-]').HasSubstructMatch(Chem.MolFromSmiles('CC[O-]'))
True
>>> Chem.MolFromSmiles('CC[O-]').HasSubstructMatch(Chem.MolFromSmiles('CC[OH]'))
True
>>> Chem.MolFromSmiles('CCOC').HasSubstructMatch(Chem.MolFromSmiles('CC[OH]'))
True
>>> Chem.MolFromSmiles('CCOC').HasSubstructMatch(Chem.MolFromSmiles('CCO'))
True
>>> Chem.MolFromSmiles('CCC').HasSubstructMatch(Chem.MolFromSmiles('CCC'))
True
>>> Chem.MolFromSmiles('CC[14C]').HasSubstructMatch(Chem.MolFromSmiles('CCC'))
True
>>> Chem.MolFromSmiles('CCC').HasSubstructMatch(Chem.MolFromSmiles('CC[14C]'))
False
>>> Chem.MolFromSmiles('CC[14C]').HasSubstructMatch(Chem.MolFromSmiles('CC[14C]'))
True
>>> Chem.MolFromSmiles('OCO').HasSubstructMatch(Chem.MolFromSmiles('C'))
True
>>> Chem.MolFromSmiles('OCO').HasSubstructMatch(Chem.MolFromSmiles('[CH]'))
False
>>> Chem.MolFromSmiles('OCO').HasSubstructMatch(Chem.MolFromSmiles('[CH2]'))
False
>>> Chem.MolFromSmiles('OCO').HasSubstructMatch(Chem.MolFromSmiles('[CH3]'))
False
>>> Chem.MolFromSmiles('OCO').HasSubstructMatch(Chem.MolFromSmiles('O[CH3]'))
True
>>> Chem.MolFromSmiles('O[CH2]O').HasSubstructMatch(Chem.MolFromSmiles('C'))
True
>>> Chem.MolFromSmiles('O[CH2]O').HasSubstructMatch(Chem.MolFromSmiles('[CH2]'))
False
```

Molecular Sanitization

The molecule parsing functions all, by default, perform a "sanitization" operation on the molecules read. The idea is to generate useful computed properties (like hybridization, ring membership, etc.) for the rest of the code and to ensure that the molecules are "reasonable": that they can be represented with octet-complete Lewis dot structures.

Here are the steps involved, in order.

1. **`clearComputedProps`: removes any computed properties that already exist** on the molecule and its atoms and bonds. This step is always performed.
2. `cleanUp`: standardizes a small number of non-standard valence states. The clean up operations are:
   - Neutral 5 valent Ns with double bonds to Os are converted to the zwitterionic form. Example: `N(=O)=O -> [N+](=O)O-]`
   - Neutral 5 valent Ns with triple bonds to another N are converted to the zwitterionic form. Example: `C-N=N#N -> C-N=[N+]=[N-]`

   - Neutral 5 valent phosphorus with one double bond to an O and another to either a C or a P are converted to the zwitterionic form. Example: `C=P(=O)O -> C=[P+]([O-])O`
   - Neutral Cl, Br, or I with exclusively O neighbors, and a valence of 3, 5, or 7, are converted to the zwitterionic form. This covers things like chlorous acid, chloric acid, and perchloric acid. Example: `O=Cl(=O)O -> [O-][Cl+2][O-]O`

   This step should not generate execptions.

3. `updatePropertyCache`: calculates the explicit and implicit valences on all atoms. This generates exceptions for atoms in higher-than-allowed valence states. This step is always performed, but if it is "skipped" the test for non-standard valences will not be carried out.
4. `symmetrizeSSSR`: calls the symmetrized smallest set of smallest rings algorithm (discussed in the Getting Started document).
5. `Kekulize`: converts aromatic rings to their Kekule form. Will raise an exception if a ring cannot be kekulized or if aromatic bonds are found outside of rings.
6. `assignRadicals`: determines the number of radical electrons (if any) on each atom.
7. `setAromaticity`: identifies the aromatic rings and ring systems (see above), sets the aromatic flag on atoms and bonds, sets bond orders to aromatic.
8. `setConjugation`: identifies which bonds are conjugated
9. `setHybridization`: calculates the hybridization state of each atom
10. `cleanupChirality`: removes chiral tags from atoms that are not sp3 hybridized.
11. `adjustHs`: adds explicit Hs where necessary to preserve the chemistry. This is typically needed for heteroatoms in aromatic rings. The classic example is the nitrogen atom in pyrrole.

The individual steps can be toggled on or off when calling `MolOps::sanitizeMol` or `Chem.SanitizeMol`.

Implementation Details

"Magic" Property Values

The following property values are regularly used in the RDKit codebase and may be useful to client code.

ROMol (Mol in Python)

Property Name	Use
MolFileComments	Read from/written to the comment line of CTABs.
MolFileInfo	Read from/written to the info line of CTABs.
_MolFileChiralFlag	Read from/written to the chiral flag of CTABs.
_Name	Read from/written to the name line of CTABs.
_smilesAtomOutputOrder	The order in which atoms were written to SMILES

Atom

Property Name	Use
_CIPCode	the CIP code (R or S) of the atom
_CIPRank	the integer CIP rank of the atom
_ChiralityPossible	set if an atom is a possible chiral center
_MolFileRLabel	integer R group label for an atom, read from/written to CTABs.
_ReactionDegreeChanged	set on an atom in a product template of a reaction if its degree changes in the reaction
_protected	atoms with this property set will not be considered as matching reactant queries in reactions
dummyLabel	(on dummy atoms) read from/written to CTABs as the atom symbol
molAtomMapNumber	the atom map number for an atom, read from/written to SMILES and CTABs
molfileAlias	the mol file alias for an atom (follows A tags), read from/written to CTABs
molFileValue	the mol file value for an atom (follows V tags), read from/written to CTABs
molFileInversionFlag	used to flag whether stereochemistry at an atom changes in a reaction, read from/written to CTABs, determined automatically from SMILES
molRxnComponent	which component of a reaction an atom belongs to, read from/written to CTABs
molRxnRole	which role an atom plays in a reaction (1=Reactant, 2=Product, 3=Agent), read from/written to CTABs
smilesSymbol	determines the symbol that will be written to a SMILES for the atom

Thread safety and the RDKit

While writing the RDKit, we did attempt to ensure that the code would work in a multi-threaded environment by avoiding use of global variables, etc. However, making code thread safe is not a completely trivial thing, so there are no doubt some gaps. This section describes which pieces of the code base have explicitly been tested for thread safety.

Note: With the exception of the small number of methods/functions that take a `numThreads` argument, this section does not apply to using the RDKit from Python threads. Boost.Python ensures that only one thread is calling into the C++ code at any point. To get concurrent execution in Python, use the multiprocessing module or one of the other standard python approaches for this .

What has been tested

- Reading molecules from SMILES/SMARTS/Mol blocks
- Writing molecules to SMILES/SMARTS/Mol blocks
- Generating 2D coordinates
- Generating 3D conformations with the distance geometry code
- Optimizing molecules with UFF or MMFF
- Generating fingerprints
- The descriptor calculators in $RDBASE/Code/GraphMol/Descriptors
- Substructure searching (Note: if a query molecule contains recursive queries, it may not be safe to use it concurrently on multiple threads, see below)
- The Subgraph code
- The ChemTransforms code

- The chemical reactions code
- The Open3DAlign code
- The MolDraw2D drawing code

Known Problems

- InChI generation and (probably) parsing. This seems to be a limitation of the IUPAC InChI code. In order to allow the code to be used in a multi-threaded environment, a mutex is used to ensure that only one thread is using the IUPAC code at a time. This is only enabled if the RDKit is built with the `RDK_TEST_MULTITHREADED` option enabled.
- The MolSuppliers (e.g. SDMolSupplier, SmilesMolSupplier?) change their internal state when a molecule is read. It is not safe to use one supplier on more than one thread.
- Substructure searching using query molecules that include recursive queries. The recursive queries modify their internal state when a search is run, so it's not safe to use the same query concurrently on multiple threads. If the code is built using the `RDK_BUILD_THREADSAFE_SSS` argument (the default for the binaries we provide), a mutex is used to ensure that only one thread is using a given recursive query at a time.

License

The intent of this license is similar to that of the RDKit itself. In simple words: "Do whatever you want with it, but please give us some credit."

CHAPTER 5

RDKit Cookbook

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.

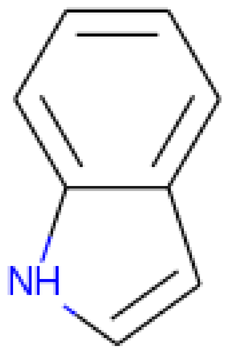
If you find mistakes, or have suggestions for improvements, please either fix them yourselves in the source document (the .rst file) or send them to the mailing list: rdkit-discuss@lists.sourceforge.net (you will need to subscribe first)

Miscellaneous Topics

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_KEKULIZE^Chem.SanitizeFlags.SANITIZE_SETAROMATICITY)
  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.

Manipulating Molecules

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:

Examples of using it:

This produces:

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()
```

Neutralizing Charged Molecules

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

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
```

3D functionality in the RDKit

The RDKit contains a range of 3D functionalities such as:

- Shape alignment
- RMS calculation
- Shape Tanimoto Distance
- Shape Protrude Distance
- 3D pharmacophore fingerprint
- Torsion fingerprint (deviation)

There are two alignment methods currently available in the RDKit. As an example we use two crystal structures from the PDB of the same molecule.

The code:

```
from rdkit import Chem, RDConfig
from rdkit.Chem import AllChem, rdMolAlign
# The reference molecule
ref = Chem.MolFromSmiles(
↪'NC(=[NH2+])c1ccc(C[C@@H](NC(=O)CNS(=O)(=O)c2ccc3ccccc3c2)C(=O)N2CCCCC2)cc1')
# The PDB conformations
mol1 = Chem.MolFromPDBFile(RDConfig.RDBaseDir+'/rdkit/Chem/test_data/1DWD_ligand.pdb')
mol1 = AllChem.AssignBondOrdersFromTemplate(ref, mol1)
mol2 = Chem.MolFromPDBFile(RDConfig.RDBaseDir+'/rdkit/Chem/test_data/1PPC_ligand.pdb')
mol2 = AllChem.AssignBondOrdersFromTemplate(ref, mol2)
# Align them
rms = rdMolAlign.AlignMol(mol1, mol2)
print(rms)
# Align them with OPEN3DAlign
pyO3A = rdMolAlign.GetO3A(mol1, mol2)
score = pyO3A.Align()
print(score)
```

This produces:

```
1.55001955728
0.376459885045
```

If a molecule contains more than one conformer, they can be aligned with respect to the first conformer. If a list is provided to the option RMSlist, the RMS value from the alignment are stored. The RMS value of two conformers of a molecule can also be calculated separately, either with or without alignment (using the flag prealigned).

Examples of using it:

```
from rdkit import Chem
from rdkit.Chem import AllChem
mol = Chem.MolFromSmiles(
↪'NC(=[NH2+])c1ccc(C[C@@H](NC(=O)CNS(=O)(=O)c2ccc3ccccc3c2)C(=O)N2CCCCC2)cc1')
cids = AllChem.EmbedMultipleConfs(mol, numConfs=50, maxAttempts=1000,␣
↪pruneRmsThresh=0.1)
print(len(cids))
# align the conformers
rmslist = []
AllChem.AlignMolConformers(mol, RMSlist=rmslist)
print(len(rmslist))
# calculate RMS of confomers 1 and 9 separately
rms = AllChem.GetConformerRMS(mol, 1, 9, prealigned=True)
```

This produces:

```
50
49
```

For shape comparison, the RDKit provides two Shape-based distances that can be calculated for two prealigned molecules or conformers. Shape protrude distance focusses on the volume mismatch, while Shape Tanimoto distance takes the volume overlay overall into account.

Examples of using it:

```
from rdkit import Chem, RDConfig
from rdkit.Chem import AllChem, rdMolAlign, rdShapeHelpers
ref = Chem.MolFromSmiles(
↪'NC(=[NH2+])c1ccc(C[C@@H](NC(=O)CNS(=O)(=O)c2ccc3ccccc3c2)C(=O)N2CCCCC2)cc1')
```

```
mol1 = Chem.MolFromPDBFile(RDConfig.RDBaseDir+'/rdkit/Chem/test_data/1DWD_ligand.pdb')
mol1 = AllChem.AssignBondOrdersFromTemplate(ref, mol1)
mol2 = Chem.MolFromPDBFile(RDConfig.RDBaseDir+'/rdkit/Chem/test_data/1PPC_ligand.pdb')
mol2 = AllChem.AssignBondOrdersFromTemplate(ref, mol2)
rms = rdMolAlign.AlignMol(mol1, mol2)
tani = rdShapeHelpers.ShapeTanimotoDist(mol1, mol2)
prtr = rdShapeHelpers.ShapeProtrudeDist(mol1, mol2)
print(rms, tani, prtr)
```

This produces:

```
1.55001955728 0.18069102331 0.0962800875274
```

A 3D pharmacophore fingerprint can be calculated using the RDKit by feeding a 3D distance matrix to the 2D-pharmacophore machinery.

Examples of using it:

```
from rdkit import Chem, DataStructs, RDConfig
from rdkit.Chem import AllChem
from rdkit.Chem.Pharm2D import Gobbi_Pharm2D, Generate
ref = Chem.MolFromSmiles(
↪'NC(=[NH2+])c1ccc(C[C@@H](NC(=O)CNS(=O)(=O)c2ccc3ccccc3c2)C(=O)N2CCCCC2)cc1')
mol1 = Chem.MolFromPDBFile(RDConfig.RDBaseDir+'/rdkit/Chem/test_data/1DWD_ligand.pdb')
mol1 = AllChem.AssignBondOrdersFromTemplate(ref, mol1)
mol2 = Chem.MolFromPDBFile(RDConfig.RDBaseDir+'/rdkit/Chem/test_data/1PPC_ligand.pdb')
mol2 = AllChem.AssignBondOrdersFromTemplate(ref, mol2)
# pharmacophore fingerprint
factory = Gobbi_Pharm2D.factory
fp1 = Generate.Gen2DFingerprint(mol1, factory, dMat=Chem.Get3DDistanceMatrix(mol1))
fp2 = Generate.Gen2DFingerprint(mol2, factory, dMat=Chem.Get3DDistanceMatrix(mol2))
# Tanimoto similarity
tani = DataStructs.TanimotoSimilarity(fp1, fp2)
print(tani)
```

This produces:

```
0.451665312754
```

The RDKit provides an implementation of the torsion fingerprint deviation (TFD) approach developed by Schulz-Gasch et al. (J. Chem. Inf. Model, 52, 1499, 2012). For a pair of conformations of a molecule, the torsional angles of the rotatable bonds and the ring systems are recorded in a torsion fingerprint (TF), and the deviations between the TFs calculated, normalized and summed up. For each torsion, a set of four atoms a-b-c-d are selected.

The RDKit implementation allows the user to customize the torsion fingerprints as described in the following.

- In the original approach, the torsions are weighted based on their distance to the center of the molecule. By default, this weighting is performed, but can be turned off using the flag useWeights=False
- If symmetric atoms a and/or d exist, all possible torsional angles are calculated. To determine if two atoms are symmetric, the hash codes from the Morgan algorithm at a given radius are used (default: radius = 2).
- In the original approach, the maximal deviation used for normalization is 180.0 degrees for all torsions (default). If maxDev='spec', a torsion-type dependent maximal deviation is used for the normalization.
- In the original approach, single bonds adjacent to triple bonds and allenes are ignored (default). If ignoreColinearBonds='False', a "combined" torsion is used

In addition there are a few differences to the implementation by Schulz-Gasch et al.:

- Hydrogens are never considered.
- In the original approach, atoms a and/or d are chosen randomly if atom b and/or c have multiple non-symmetric neighbors. The RDKit implementation picks the atom with the smallest Morgan invariant. This way the choice is independent of the atom order in the molecule.
- In the case of symmetric atoms a and/or d, the RDKit implementation stores all possible torsional angles in the TF instead of only storing the smallest one as in the original approach. Subsequently, all possible deviations are determined and the smallest one used for the TFD calculation. This procedure guarantees that the smallest deviations enter the TFD.

Examples of using it:

```
from rdkit import Chem, RDConfig
from rdkit.Chem import AllChem, TorsionFingerprints
ref = Chem.MolFromSmiles(
↪'NC(=[NH2+])c1ccc(C[C@@H](NC(=O)CNS(=O)(=O)c2ccc3ccccc3c2)C(=O)N2CCCCC2)cc1')
mol1 = Chem.MolFromPDBFile(RDConfig.RDBaseDir+'/rdkit/Chem/test_data/1DWD_ligand.pdb')
mol1 = AllChem.AssignBondOrdersFromTemplate(ref, mol1)
mol2 = Chem.MolFromPDBFile(RDConfig.RDBaseDir+'/rdkit/Chem/test_data/1PPC_ligand.pdb')
mol2 = AllChem.AssignBondOrdersFromTemplate(ref, mol2)
tfd1 = TorsionFingerprints.GetTFDBetweenMolecules(mol1, mol2)
tfd2 = TorsionFingerprints.GetTFDBetweenMolecules(mol1, mol2, useWeights=False)
tfd3 = TorsionFingerprints.GetTFDBetweenMolecules(mol1, mol2, maxDev='spec')
print(tfd1, tfd2, tfd3)
```

This produces:

```
0.0691236990428 0.111475253992 0.0716255058804
```

If the TFD between conformers of the same molecule is to be calculated, the function GetTFDBetweenConformers() should be used for performance reasons.

Examples of using it:

```
from rdkit import Chem, RDConfig
from rdkit.Chem import AllChem, TorsionFingerprints
ref = Chem.MolFromSmiles(
↪'NC(=[NH2+])c1ccc(C[C@@H](NC(=O)CNS(=O)(=O)c2ccc3ccccc3c2)C(=O)N2CCCCC2)cc1')
mol1 = Chem.MolFromPDBFile(RDConfig.RDBaseDir+'/rdkit/Chem/test_data/1DWD_ligand.pdb')
mol1 = AllChem.AssignBondOrdersFromTemplate(ref, mol1)
mol2 = Chem.MolFromPDBFile(RDConfig.RDBaseDir+'/rdkit/Chem/test_data/1PPC_ligand.pdb')
mol1.AddConformer(mol2.GetConformer(), assignId=True)
tfd = TorsionFingerprints.GetTFDBetweenConformers(mol1, confIds1=[0], confIds2=[1])
print(tfd)
```

This produces:

```
[0.0691...]
```

For the conformer RMS and TFD values, the RDKit provides convenience functions that calculated directly the symmetric matrix which can be fed into a clustering algorithm such as Butina clustering. The flag reordering ensures that the number of neighbors of the unclustered molecules is updated every time a cluster is created.

Examples of using it:

```
from rdkit import Chem
from rdkit.Chem import AllChem, TorsionFingerprints
from rdkit.ML.Cluster import Butina
mol = Chem.MolFromSmiles(
↪'NC(=[NH2+])c1ccc(C[C@@H](NC(=O)CNS(=O)(=O)c2ccc3ccccc3c2)C(=O)N2CCCCC2)cc1')
```

```
cids = AllChem.EmbedMultipleConfs(mol, numConfs=50, maxAttempts=1000,␣
↪pruneRmsThresh=0.1)
# RMS matrix
rmsmat = AllChem.GetConformerRMSMatrix(mol, prealigned=False)
# TFD matrix
tfdmat = TorsionFingerprints.GetTFDMatrix(mol)
# clustering
num = mol.GetNumConformers()
rms_clusters = Butina.ClusterData(rmsmat, num, 2.0, isDistData=True, reordering=True)
tfd_clusters = Butina.ClusterData(tfdmat, num, 0.3, isDistData=True, reordering=True)
```

Using scikit-learn with RDKit

scikit-learn is a machine-learning library for Python containing a variety of supervised and unsupervised methods. The documention can be found here: http://scikit-learn.org/stable/user_guide.html

RDKit fingerprints can be used to train machine-learning models from scikit-learn. Here is an example for random forest:

The code:

```
from rdkit import Chem, DataStructs
from rdkit.Chem import AllChem
from sklearn.ensemble import RandomForestClassifier
import numpy

# generate four molecules
m1 = Chem.MolFromSmiles('c1ccccc1')
m2 = Chem.MolFromSmiles('c1ccccc1CC')
m3 = Chem.MolFromSmiles('c1ccncc1')
m4 = Chem.MolFromSmiles('c1ccncc1CC')
mols = [m1, m2, m3, m4]

# generate fingeprints: Morgan fingerprint with radius 2
fps = [AllChem.GetMorganFingerprintAsBitVect(m, 2) for m in mols]

# convert the RDKit explicit vectors into numpy arrays
np_fps = []
for fp in fps:
  arr = numpy.zeros((1,))
  DataStructs.ConvertToNumpyArray(fp, arr)
  np_fps.append(arr)

# get a random forest classifiert with 100 trees
rf = RandomForestClassifier(n_estimators=100, random_state=1123)

# train the random forest
# with the first two molecules being actives (class 1) and
# the last two being inactives (class 0)
ys_fit = [1, 1, 0, 0]
rf.fit(np_fps, ys_fit)

# use the random forest to predict a new molecule
m5 = Chem.MolFromSmiles('c1ccccc1O')
fp = numpy.zeros((1,))
DataStructs.ConvertToNumpyArray(AllChem.GetMorganFingerprintAsBitVect(m5, 2), fp)
```

```
print(rf.predict((fp,)))
print(rf.predict_proba((fp,)))
```

The output with scikit-learn version 0.13 is:

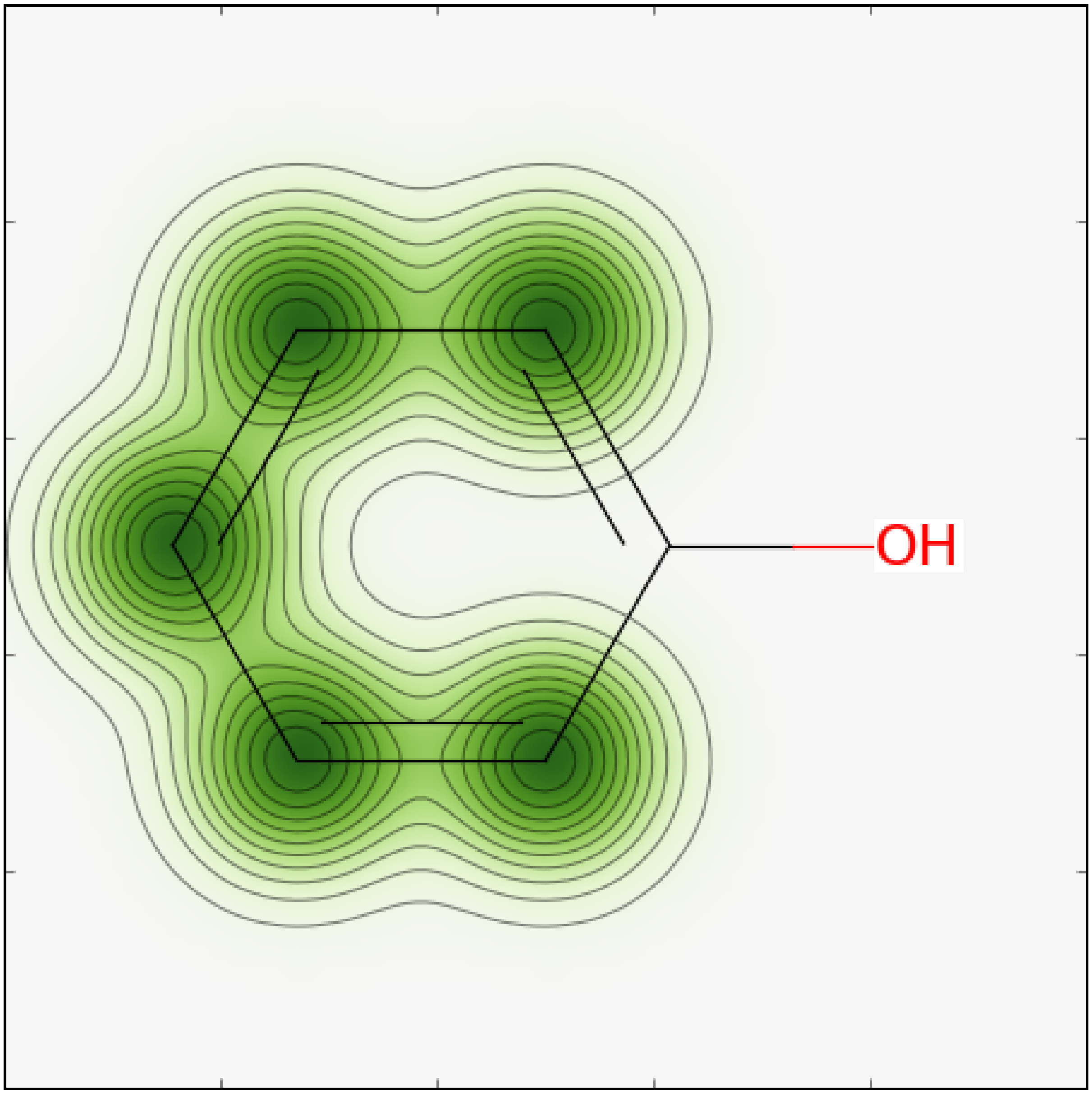
```
[1]
[[ 0.14 0.86]]
```

Generating a similarity map for this model.

The code:

```
from rdkit.Chem.Draw import SimilarityMaps

# helper function
def getProba(fp, predictionFunction):
  return predictionFunction((fp,))[0][1]

m5 = Chem.MolFromSmiles('c1ccccc1O')
fig, maxweight = SimilarityMaps.GetSimilarityMapForModel(m5, SimilarityMaps.
↪GetMorganFingerprint, lambda x: getProba(x, rf.predict_proba))
```

This produces:



Using custom MCS atom types

Mailing list discussion: http://www.mail-archive.com/rdkit-discuss@lists.sourceforge.net/msg03676.html

IPython notebook: http://nbviewer.ipython.org/gist/greglandrum/8351725 https://gist.github.com/greglandrum/8351725

The goal is to be able to use custom atom types in the MCS code, yet still be able to get a readable SMILES for the MCS. We will use the MCS code's option to use isotope information in the matching and then set bogus isotope values that contain our isotope information.

The code:

```
from rdkit import Chem
from rdkit.Chem import rdFMCS

# our test molecules:
smis=["COc1ccc(C(Nc2nc3c(ncn3COCC=O)c(=O)[nH]2)(c2ccccc2)c2ccccc2)cc1",
      "COc1ccc(C(Nc2nc3c(ncn3COC(CO)(CO)CO)c(=O)[nH]2)(c2ccccc2)c2ccccc2)cc1"]
ms = [Chem.MolFromSmiles(x) for x in smis]

def label(a):
  " a simple hash combining atom number and hybridization "
  return 100*int(a.GetHybridization())+a.GetAtomicNum()

# copy the molecules, since we will be changing them
nms = [Chem.Mol(x) for x in ms]
for nm in nms:
  for at in nm.GetAtoms():
      at.SetIsotope(label(at))

mcs=rdFMCS.FindMCS(nms,atomCompare=rdFMCS.AtomCompare.CompareIsotopes)
print(mcs.smartsString)
```

This generates the following output:

```
[406*]-[308*]-[306*]1:[306*]:[306*]:[306*](:[306*]:[306*]:1)-[406*](-[307*]-
↪[306*]1:[307*]:[306*]2:[306*](:[306*](:[307*]:1)=[308*]):[307*]:[306*]:[307*]:2-
↪[406*]-[408*]-[406*])(-[306*]1:[306*]:[306*]:[306*]:[306*]:[306*]:1)-
↪[306*]1:[306*]:[306*]:[306*]:[306*]:[306*]:1
```

That's what we asked for, but it's not exactly readable. We can get to a more readable form in a two step process:

1. Do a substructure match of the MCS onto a copied molecule
2. Generate SMILES for the original molecule, using only the atoms that matched in the copy.

This works because we know that the atom indices in the copies and the original molecules are the same.

```
def getMCSSmiles(mol,labelledMol,mcs):
    mcsp = Chem.MolFromSmarts(mcs.smartsString)
    match = labelledMol.GetSubstructMatch(mcsp)
    return Chem.MolFragmentToSmiles(mol,atomsToUse=match,
                                    isomericSmiles=True,
                                    canonical=False)

print(getMCSSmiles(ms[0],nms[0],mcs))

COc1ccc(C(Nc2nc3c(ncn3COC)c(=O)[nH]2)(c2ccccc2)c2ccccc2)cc1
```

That's what we were looking for.

Clustering molecules

For large sets of molecules (more than 1000-2000), it's most efficient to use the Butina clustering algorithm.

Here's some code for doing that for a set of fingerprints:

```
def ClusterFps(fps,cutoff=0.2):
    from rdkit import DataStructs
    from rdkit.ML.Cluster import Butina

    # first generate the distance matrix:
    dists = []
    nfps = len(fps)
    for i in range(1,nfps):
        sims = DataStructs.BulkTanimotoSimilarity(fps[i],fps[:i])
        dists.extend([1-x for x in sims])

    # now cluster the data:
    cs = Butina.ClusterData(dists,nfps,cutoff,isDistData=True)
    return cs
```

The return value is a tuple of clusters, where each cluster is a tuple of ids.

Example usage:

```
from rdkit import Chem
from rdkit.Chem import AllChem
import gzip
ms = [x for x in Chem.ForwardSDMolSupplier(gzip.open('zdd.sdf.gz')) if x is not None]
fps = [AllChem.GetMorganFingerprintAsBitVect(x,2,1024) for x in ms]
clusters=ClusterFps(fps,cutoff=0.4)
```

The variable clusters contains the results:

```
>>> print(clusters[200])
(6164, 1400, 1403, 1537, 1543, 6575, 6759)
```

That cluster contains 7 points, the centroid is point 6164.

RMSD Calculation between N molecules

Introduction

We sometimes need to calculate RMSD distances between two (or more) molecules. This can be used to calculate how close two conformers are. Most RMSD calculations make sense only on similar compounds or, at least, for common parts in different compounds.

Details

The following program (written in python 2.7) takes an SDF file as an input and generates all the RMSD distances between the molecules in that file. These distances are written to an output file (user defined).

So for an SDF with 5 conformers we will get 10 RMSD scores - typical n choose k problem, without repetition i.e. 5! / 2!(5-2)!

The code:

```
#!/usr/bin/python
'''
calculates RMSD differences between all structures in a file

@author: JP <jp@javaclass.co.uk>
'''
```

```
import os
import getopt
import sys

# rdkit imports
from rdkit import Chem
from rdkit.Chem import AllChem

'''
Write contents of a string to file
'''
def write_contents(filename, contents):
  # do some basic checking, could use assert strictly speaking
  assert filename is not None, "filename cannot be None"
  assert contents is not None, "contents cannot be None"
  f = open(filename, "w")
  f.write(contents)
  f.close() # close the file

'''
Write a list to a file
'''
def write_list_to_file(filename, list, line_sep = os.linesep):
  # do some basic checking, could use assert strictly speaking
  assert list is not None and len(list) > 0, "list cannot be None or empty"
  write_contents(filename, line_sep.join(list))

'''
Calculate RMSD spread
'''
def calculate_spread(molecules_file):

  assert os.path.isfile(molecules_file), "File %s does not exist!" % molecules

  # get an iterator
  mols = Chem.SDMolSupplier(molecules_file)

  spread_values = []
  # how many molecules do we have in our file
  mol_count = len(mols)
  # we are going to compare each molecule with every other molecule
  # typical n choose k scenario (n choose 2)
  # where number of combinations is given by (n!) / k!(n-k)! ; if my maths isn't too␣
↪rusty
  for i in range(mol_count - 1):
      for j in range(i+1, mol_count):
          # show something is being done ... because for large mol_count this will␣
↪take some time
          print("Aligning molecule #%d with molecule #%d (%d molecules in all)" % (i,␣
↪j, mol_count))
          # calculate RMSD and store in an array
          # unlike AlignMol this takes care of symmetry
          spread_values.append(str(AllChem.GetBestRMS(mols[i], mols[j])))
  # return that array
  return spread_values


def main():
  try:
      # the options are as follows:
      # f - the actual structure file
      opts, args = getopt.getopt(sys.argv[1:], "vf:o:")
  except getopt.GetoptError, err:
      # print help information and exit:
      print(str(err)) # will print something like "option -a not recognized"
      sys.exit(401)

  # DEFAULTS
  molecules_file  = None
  output_file = None

  for opt, arg in opts:
      if opt == "-v":
          print("RMSD Spread 1.1")
          sys.exit()
      elif opt == "-f":
          molecules_file = arg
      elif opt == "-o":
          output_file = arg
      else:
          assert False, "Unhandled option: " + opt

  # assert the following - not the cleanest way to do this but this will work
  assert molecules_file is not None, "file containing molecules must be specified,␣
↪add -f to command line arguments"
  assert output_file is not None, "output file must be specified, add -o to command␣
↪line arguments"
  # get the RMSD spread values
  spread_values = calculate_spread(molecules_file)
  # write them to file
  write_list_to_file(output_file, spread_values)



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

This program may be executed at the command line in the following manner (provided you have your python interpreter at `/usr/bin/python`, otherwise edit the first line; the funnily named shebang):

```
calculate_spread.py -f my_conformers.sdf -o my_conformers.rmsd_spread.txt
```

TL;DR : The line `AllChem.GetBestRMS(mol1, mol2)` returns the RMSD as a float and is the gist of this program. `GetBestRMS()` takes care of symmetry unlike `AlignMol()`

License

please give us some credit.”

CHAPTER 6

The RDKit database cartridge

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 .md file) or send them to the mailing list: rdkit-discuss@lists.sourceforge.net (you will need to subscribe first)

Tutorial

Introduction

Creating databases

Configuration

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 :

```
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
```

Creating a database from a file

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.

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 integer, parent_id integer)' emolecules
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 "copy raw_data (smiles,emol_id,parent_id) from stdin with␣
↪delimiter ' '" emolecules
```

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 where m is not null;
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.

Loading ChEMBL

Start by downloading and installing the postgresql dump from the ChEMBL website ftp://ftp.ebi.ac.uk/pub/databases/chembl/ChEMBLdb/latest

Connect to the database, install the cartridge, and create the schema that we'll use:

```
chembl_14=# create extension if not exists rdkit;
chembl_14=# create schema rdk;
```

Create the molecules and build the substructure search index:

```
chembl_14=# select * into rdk.mols from (select molregno,mol_from_
↪ctab(molfile::cstring) m  from compound_structures) tmp where m is not null;
SELECT 1210823
```

```
chembl_14=# create index molidx on rdk.mols using gist(m);
CREATE INDEX
chembl_14=# alter table rdk.mols add primary key (molregno);
NOTICE:  ALTER TABLE / ADD PRIMARY KEY will create implicit index "mols_pkey" for␣
↪table "mols"
ALTER TABLE
```

Create some fingerprints and build the similarity search index:

```
chembl_14=# select molregno,torsionbv_fp(m) as torsionbv,morganbv_fp(m) as mfp2,
↪featmorganbv_fp(m) as ffp2 into rdk.fps from rdk.mols;
SELECT 1210823
chembl_14=# create index fps_ttbv_idx on rdk.fps using gist(torsionbv);
CREATE INDEX
chembl_14=# create index fps_mfp2_idx on rdk.fps using gist(mfp2);
CREATE INDEX
chembl_14=# create index fps_ffp2_idx on rdk.fps using gist(ffp2);
CREATE INDEX
chembl_14=# alter table rdk.fps add primary key (molregno);
NOTICE:  ALTER TABLE / ADD PRIMARY KEY will create implicit index "fps_pkey" for␣
↪table "fps"
ALTER TABLE
```

Here is a group of the commands used here (and below) in one block so that you can just paste it in at the psql prompt:

```
create extension if not exists rdkit;
create schema rdk;
select * into rdk.mols from (select molregno,mol_from_ctab(molfile::cstring) m  from␣
↪compound_structures) tmp where m is not null;
create index molidx on rdk.mols using gist(m);
alter table rdk.mols add primary key (molregno);
select molregno,torsionbv_fp(m) as torsionbv,morganbv_fp(m) as mfp2,featmorganbv_
↪fp(m) as ffp2 into rdk.fps from rdk.mols;
create index fps_ttbv_idx on rdk.fps using gist(torsionbv);
create index fps_mfp2_idx on rdk.fps using gist(mfp2);
create index fps_ffp2_idx on rdk.fps using gist(ffp2);
alter table rdk.fps add primary key (molregno);
create or replace function get_mfp2_neighbors(smiles text)
returns table(molregno integer, m mol, similarity double precision) as
```

$$ select molregno,m,tanimoto_sml(morganbv_fp(mol_from_smiles($1::cstring)),mfp2) as similarity from rdk.fps join rdk.mols using (molregno) where morganbv_fp(mol_from_smiles($1::cstring))%mfp2 order by morganbv_fp(mol_from_smiles($1::cstring))<%>mfp2; $$ language sql stable ;

Substructure searches

Example query molecules taken from the eMolecules home page:

```
chembl_14=# select count(*) from rdk.mols where m@>'c1cccc2c1nncc2' ;
 count
-------
   281
(1 row)

Time: 184.043 ms
chembl_14=# select count(*) from rdk.mols where m@>'c1ccnc2c1nccn2' ;
```

```
 count
-------
   671
(1 row)

Time: 449.998 ms
chembl_14=# select count(*) from rdk.mols where m@>'c1cncc2n1ccn2' ;
 count
-------
   930
(1 row)

Time: 568.378 ms
chembl_14=# select count(*) from rdk.mols where m@>'Nc1ncnc(N)n1' ;
 count
-------
  4478
(1 row)

Time: 721.758 ms
chembl_14=# select count(*) from rdk.mols where m@>'c1scnn1' ;
 count
-------
 10908
(1 row)

Time: 701.036 ms
chembl_14=# select count(*) from rdk.mols where m@>'c1cccc2c1ncs2' ;
 count
-------
 12823
(1 row)

Time: 1585.473 ms
chembl_14=# select count(*) from rdk.mols where m@>'c1cccc2c1CNCCN2' ;
 count
-------
  1155
(1 row)

Time: 4567.222 ms
```

Notice that the last two queries are starting to take a while to execute and count all the results.

Given we're searching through 1.2 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:

```
chembl_14=# select * from rdk.mols where m@>'c1cccc2c1CNCCN2' limit 100;
 molregno |                                                                   ␣
↪            m
----------+--------------------------------------------------------------------
↪-----------------------------------------------------------------------------
↪---------------
  1292129 | Cc1ccc2c(c1)C(=O)N(N(C)C)CC(=O)N2
  1013311 | CCCCC(=O)N1CC(=O)Nc2ccc(F)cc2C1c1ccccc1
  1294754 | COc1cc2c(cc1OCc1ccccc1)NC(=O)[C@@H]1CCCN1C2=O
  1012025 | O=C(c1cc2ccccc2oc1=O)N1CC(=O)Nc2ccc(Br)cc2C1c1ccc(F)cc1
   995226 | CC1Cc2ccccc2N1C(=O)CN1c2ccccc2C(=O)N(C)CC1=O
  1291875 | COC(=O)C1=NN2c3ccccc3CN([C@@H](C)c3ccccc3)C(=O)[C@@H]2[C@H]1c1ccccc1
  ...
  1116370 | COc1ccc(CC(=O)N2CC(=O)Nc3ccc(Br)cc3C2c2ccc(F)cc2)cc1OC
  1114872 |␣
↪O=C1[C@@H]2[C@H](C(=O)N1Cc1ccccc1)[C@@H]1C(=O)Nc3ccccc3C(=O)N1[C@@H]2c1ccccc1
Time: 375.747 ms
```

SMARTS-based queries

Oxadiazole or thiadiazole:

```
chembl_14=# select * from rdk.mols where m@>'c1[o,s]ncn1'::qmol limit 500;
 molregno |                                                                    m
----------+-----------------------------------------------------------------------
↪-----------------------------------------------------------------
   534296 | Clc1ccccc1CNc1noc(-c2sccc2Br)n1
     1178 | CCCCc1oc2ccccc2c1Cc1cccc(/C(C)=C/Cn2oc(=O)[nH]c2=O)c1
   566382 | COC(=O)CCc1nc(C2CC(c3ccc(O)c(F)c3)=NO2)no1
   499261 | CS/C=C(/C)n1c(=O)onc1C(=O)c1ccc(Br)cc1
   450499 | CS(=O)(=O)c1ccc(Nc2ncnc(N3CCC(c4nc(-
↪c5cccc(C(F)(F)F)c5)no4)CC3)c2[N+](=O)[O-])cc1
   600176 | Cc1nc(-c2c(Cl)cc(Cl)cc2-c2cnc([C@@H](C)NC(=O)N(C)O)c(F)c2)no1
     1213 | CC/C(=C\Cn1oc(=O)[nH]c1=O)c1cccc(OCc2nc(-c3ccc(C(F)(F)F)cc3)oc2C)c1
   659277 | Cn1c(N)c(CCCN)c[n+]1CC1=C(C(=O)O)N2C(=O)[C@@H](NC(=O)/
↪C(=N\OC(C)(C)C(=O)O)c3nsc(N)n3)[C@H]2SC1
     1316 | CCCCCCCC/C(=C\Cn1oc(=O)[nH]c1=O)c1cccc(OCc2nc(-c3ccc(C(F)(F)F)cc3)oc2C)c1
   ...
     1206 | C/C(Cn1oc(=O)[nH]c1=O)=C(/C)c1cccc(OCc2nc(-c3ccc(C(F)(F)F)cc3)oc2C)c1
     1496 | Cc1oc(-c2ccccc2)nc1COc1cccc(C#CC(C)n2oc(=O)[nH]c2=O)c1
Time: 3365.309 ms
```

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

Using Stereochemistry

Note that by default stereochemistry is not taken into account when doing substructure queries:

```
chembl_14=# select * from rdk.mols where m@>'NC(=O)[C@@H]1CCCN1C=O' limit 10;
 molregno |                                                                       ␣
↪                m
----------+-----------------------------------------------------------------------
↪-----------------------------------------------------------------------------
↪-------------------
  1295889 |␣
↪COc1ccc(C[C@@H](C(=O)NCC(N)=O)N(C)C(=O)[C@@H]2CCCN2C(=O)[C@H](CC(C)C)NC(=O)C(C)NC(=O)OCc2ccccc2)cc
  1293815 | CN1C(=O)C23CC4=CC=CC(O)C4N2C(=O)C1(CO)SS3
  1293919 |␣
↪CNC(=O)CNC(=O)C(NC(=O)CNC(=O)C1CCCN1C(=O)C(C)NC(=O)C(NC(=O)OC(C)(C)C)C(C)C)C(C)C
  1011887 | COC(=O)C(C)NC(=O)C1CCCN1C(=O)CNC(=O)OCc1ccccc1
  1293021 |␣
↪CCC(C)C1NC(=O)C(NC(=O)C(CC(C)C)N(C)C(=O)[C@@H]2CC(O)CN2C(=O)[C@H](C)O)C(C)OC(=O)[C@H](Cc2ccc(OC)cc
  1287353 |␣
↪CCC(C)C1NC(=O)C(NC(=O)C(CC(C)C)N(C)C(=O)C2CCCN2C(=O)C(C)O)C(C)OC(=O)C(Cc2ccc(OC)cc2)N(C)C(=O)C2CCC
```

```
  1293647 |␣
↪CCC(C)[C@@H]1NC(=O)[C@@H]2CCCN2C(=O)C(CC(O)CCl)OC(=O)CCNC(=O)[C@H](C)N(C)C(=O)[C@H](C(C)C)N(C)C1=O
  1290320 |␣
↪C=CCOC(=O)[C@@H]1C[C@@H](OC(C)(C)C)CN1C(=O)[C@@H]1[C@H]2OC(C)(C)O[C@H]2CN1C(=O)OCC1c2ccccc2-
↪c2ccccc21
  1281392 |␣
↪COC1=CC2C(=O)N(C)[C@@H](C)C(=O)N3NCCC[C@@H]3C(=O)N3[C@@H](C[C@@]4(O)c5ccc(Cl)cc5N[C@@H]34)C(=O)N[C
  1014237 | CC(C)COC(=O)N1CC(O)CC1C(=O)Nc1ccc2c(c1)OCO2
(10 rows)

Time: 9.447 ms
```

This can be changed using the rdkit.do_chiral_sss configuration variable:

```
chembl_14=# set rdkit.do_chiral_sss=true;
SET
Time: 0.241 ms
chembl_14=# select * from rdk.mols where m@>'NC(=O)[C@@H]1CCCN1C=O' limit 10;
 molregno |                                                                                   ␣
↪                                                                                           ␣
↪                                                                                           ␣
↪                                m
----------+---------------------------------------------------------------------------
↪-----------------------------------------------------------------------------------
↪-----------------------------------------------------------------------------------
↪-----------------------------------------------------------------------------------
↪-----------------------------------------------------------------------------------
↪-----------------------------------------------------------------------------------
↪---------------------------------------------------
  1295889 |␣
↪COc1ccc(C[C@@H](C(=O)NCC(N)=O)N(C)C(=O)[C@@H]2CCCN2C(=O)[C@H](CC(C)C)NC(=O)C(C)NC(=O)OCc2ccccc2)cc1
  1293021 |␣
↪CCC(C)C1NC(=O)C(NC(=O)C(CC(C)C)N(C)C(=O)[C@@H]2CC(O)CN2C(=O)[C@H](C)O)C(C)OC(=O)[C@H](Cc2ccc(OC)cc2
  1293647 |␣
↪CCC(C)[C@@H]1NC(=O)[C@@H]2CCCN2C(=O)C(CC(O)CCl)OC(=O)CCNC(=O)[C@H](C)N(C)C(=O)[C@H](C(C)C)N(C)C1=O
  1290320 |␣
↪C=CCOC(=O)[C@@H]1C[C@@H](OC(C)(C)C)CN1C(=O)[C@@H]1[C@H]2OC(C)(C)O[C@H]2CN1C(=O)OCC1c2ccccc2-
↪c2ccccc21
  1281392 |␣
↪COC1=CC2C(=O)N(C)[C@@H](C)C(=O)N3NCCC[C@@H]3C(=O)N3[C@@H](C[C@@]4(O)c5ccc(Cl)cc5N[C@@H]34)C(=O)N[C
  1007418 | C/C=C\C=C\C(=O)N1CC2(CC(c3cccc(NC(=O)/C=C\C=C/C)c3)=NO2)C[C@H]1C(N)=O
   785530 | C/C=C/C(=O)N1CC2(CC(c3cccc(NC(=O)CC)c3)=NO2)C[C@H]1C(N)=O
  1292152 |␣
↪CCCCCCCC(=O)N[C@H](C(=O)N[C@H](C(=O)N(C)[C@H](C(=O)N1CCC[C@H]1C(=O)N(C)[C@H](C)C(=O)NCc1ccc(OC)cc1C
  1281390 |␣
↪CC(C)[C@@H]1NC(=O)[C@@H]2C[C@@]3(O)c4ccc(Cl)cc4N[C@H]3N2C(=O)[C@H]2CCCNN2C(=O)[C@@H](C)N(C)C(=O)[C
  1057962 |␣
↪CC[C@H](C)[C@@H]1NC(=O)[C@H](CCCNC(=N)N)NC(=O)[C@H](CC(=O)O)NC(=O)[C@H](CCSC)NC(=O)[C@H](CCCCN)NC(=
(10 rows)

Time: 35.383 ms
```

Tuning queries

It is frequently useful to be able to exert a bit more control over substructure queries without having to construct complex SMARTS queries. The cartridge function `mol_adjust_query_properties()` can be used to do just this. Here is an example of the default behavior, using aquery for 2,6 di-substituted pyridines:

```
chembl_21=# select molregno,m from rdk.mols where m@>mol_adjust_query_properties(
↪'*c1cccc(NC(=O)*)n1') limit 10;
 molregno |                                          m
----------+-----------------------------------------------------------------------
↪---------
   562310 | COc1ccc(C2C(C(=O)Nc3cccc(C)n3)c3ccccc3C(=O)N2C2CCCCC2)cc1
  1607792 | O=C(Nc1cccc(-c2ccccc2)n1)N(CCC(c1ccccc1)c1ccccc1)CCN1CCOCC1
  1587116 | O=C(Nc1cccc(F)n1)c1cc(F)cc(Oc2cncnc2)c1
  1587131 | O=C(Nc1cccc(Cl)n1)c1cc(Cl)cc(Oc2cncnc2)c1
  1592611 |␣
↪Cc1cccc(NC(=O)[C@@H](C[C@H](O)[C@@H](N)CN2CC(=O)N(c3ccccc3Cl)CC2(C)C)C(C)C)n1
  1033789 | Cc1cccc(NC(=O)C(C)n2cc([N+](=O)[O-])cn2)n1
  1571533 | CCCOc1ccc(C(=O)Nc2cccc(CC)n2)cc1
  1592338 | CN1CCC(C(=O)c2cccc(NC(=O)c3c(F)cc(F)cc3F)n2)CC1
  1053301 | Cc1cccc(NC(=O)C(Cc2ccccc2)NS(=O)(=O)c2cccc3nsnc23)n1
  1323252 |␣
↪O=C(Nc1cccc(NC(=O)C(=O)C2CCC3=C(Sc4ccccc4N3)C2=O)n1)C(=O)C1CCC2=C(Sc3ccccc3N2)C1=O
(10 rows)
```

By default `mol_adjust_query_properties()` makes the following changes to the molecule:

- Converts dummy atoms into "any" queries
- Adds a degree query to every ring atom so that its substitution must match what was provided
- Aromaticity perception is done (if it hasn't been done already)

We can control the behavior by providing an additional JSON argument. Here's an example where we disable the additional degree queries:

```
chembl_21=# select molregno,m from rdk.mols where m@>mol_adjust_query_properties(
↪'*c1cccc(NC(=O)*)n1',
chembl_21(# '{"adjustDegree":false}') limit 10;
 molregno |                                     m
----------+-----------------------------------------------------------------------
↪-
   526211 | COc1ccc2nccc(N3CCCN(CCNCc4ccc5c(n4)NC(=O)CS5)CC3)c2n1.Cl
   531033 | COc1ccc2nccc(N3CCC(CCNCc4ccc5c(n4)NC(=O)CS5)CC3)c2c1
   527205 | COc1ccc2nccc(N3CCC(O)(CCNCc4ccc5c(n4)NC(=O)CS5)CC3)c2n1.Cl
   528941 | COc1cc(F)c2ncc(F)c(CCN3C[C@@H](O)[C@@H](CNCc4ccc5c(n4)NC(=O)CS5)C3)c2c1.Cl
   539575 | COc1cc(F)c2ncc(Cl)c([C@@H](O)CN3CCC(NCc4ccc5c(n4)NC(=O)CO5)CC3)c2c1.Cl
   542344 | COc1ccc2ncc(C#N)c(CCN3C4CCC3CC(NCc3ccc5c(n3)NC(=O)CS5)C4)c2c1
   530834 | COc1ccc2ncc(F)c(CCN3CC[C@H](NC(=O)c4ccc5c(n4)NC(=O)CS5)[C@H](O)C3)c2n1
   613181 | O=C(Nc1ccc(-c2ccncc2)c(-c2ccccn2)n1)C1CC1
   527469 | COc1ccc2nccc(N3CCC(CCNCc4ccc5c(n4)NC(=O)CO5)CC3)c2c1
   527419 | COc1ccc2ncc(C#N)c(CCN3CCC(NCc4ccc5c(n4)NC(=O)CS5)CC3)c2n1
(10 rows)
```

or where we don't add the additional degree queries to ring atoms or dummies (they are only added to chain atoms):

```
chembl_21=# select molregno,m from rdk.mols where m@>mol_adjust_query_properties(
↪'*c1cccc(NC(=O)*)n1',
chembl_21(# '{"adjustDegree":true,"adjustDegreeFlags":"IGNORERINGS|IGNOREDUMMIES"}')␣
↪limit 10;
 molregno |                                     m
----------+-----------------------------------------------------------------------
↪-
   526211 | COc1ccc2nccc(N3CCCN(CCNCc4ccc5c(n4)NC(=O)CS5)CC3)c2n1.Cl
```

```
  531033 | COc1ccc2nccc(N3CCC(CCNCc4ccc5c(n4)NC(=O)CS5)CC3)c2c1
  527205 | COc1ccc2nccc(N3CCC(O)(CCNCc4ccc5c(n4)NC(=O)CS5)CC3)c2n1.Cl
  528941 | COc1cc(F)c2ncc(F)c(CCN3C[C@@H](O)[C@@H](CNCc4ccc5c(n4)NC(=O)CS5)C3)c2c1.Cl
  539575 | COc1cc(F)c2ncc(Cl)c([C@@H](O)CN3CCC(NCc4ccc5c(n4)NC(=O)CO5)CC3)c2c1.Cl
  542344 | COc1ccc2ncc(C#N)c(CCN3C4CCC3CC(NCc3ccc5c(n3)NC(=O)CS5)C4)c2c1
  530834 | COc1ccc2ncc(F)c(CCN3CC[C@H](NC(=O)c4ccc5c(n4)NC(=O)CS5)[C@H](O)C3)c2n1
  613181 | O=C(Nc1ccc(-c2ccncc2)c(-c2ccccn2)n1)C1CC1
  527469 | COc1ccc2nccc(N3CCC(CCNCc4ccc5c(n4)NC(=O)CO5)CC3)c2c1
  527419 | COc1ccc2ncc(C#N)c(CCN3CCC(NCc4ccc5c(n4)NC(=O)CS5)CC3)c2n1
(10 rows)
```

The options available are:

- **adjustDegree** (default: true) : adds a query to match the input atomic degree
- **adjustDegreeFlags** (default: ADJUST_IGNOREDUMMIES | ADJUST_IGNORECHAINS) controls where the degree is adjusted
- **adjustRingCount** (default: false) : adds a query to match the input ring count
- **adjustRingCountFlags** (default: ADJUST_IGNOREDUMMIES | ADJUST_IGNORECHAINS) controls where the ring count is adjusted
- **makeDummiesQueries** (default: true) : convert dummy atoms in the input structure into any-atom queries
- **aromatizeIfPossible** (default: true) : run the aromaticity perception algorithm on the input structure (note: this is largely redundant since molecules built from smiles always have aromaticity perceived)
- **makeBondsGeneric** (default: false) : convert bonds into any-bond queries
- **makeBondsGenericFlags** (default: false) : controls which bonds are made generic
- **makeAtomsGeneric** (default: false) : convert atoms into any-atom queries
- **makeAtomsGenericFlags** (default: false) : controls which atoms are made generic

The various `Flags` arguments mentioned above, which control where particular options are applied, are constructed by combining operations from the list below with the `|` character.

- **IGNORENONE** : apply the operation to all atoms
- **IGNORERINGS** : do not apply the operation to ring atoms
- **IGNORECHAINS** : do not apply the operation to chain atoms
- **IGNOREDUMMIES** : do not apply the operation to dummy atoms
- **IGNORENONDUMMIES** : do not apply the operation to non-dummy atoms
- **IGNOREALL** : do not apply the operation to any atoms

Similarity searches

Basic similarity searching:

```
chembl_14=# select count(*) from rdk.fps where mfp2%morganbv_fp('Cc1ccc2nc(-
↪c3ccc(NC(C4N(C(c5cccs5)=O)CCC4)=O)cc3)sc2c1');
 count
-------
    66
(1 row)
```

```
Time: 826.886 ms
```

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

```
chembl_14=# create or replace function get_mfp2_neighbors(smiles text)
    returns table(molregno integer, m mol, similarity double precision) as
  $$
  select molregno,m,tanimoto_sml(morganbv_fp(mol_from_smiles($1::cstring)),mfp2) as␣
↪similarity
  from rdk.fps join rdk.mols using (molregno)
  where morganbv_fp(mol_from_smiles($1::cstring))%mfp2
  order by morganbv_fp(mol_from_smiles($1::cstring))<%>mfp2;
  $$ language sql stable ;
CREATE FUNCTION
Time: 0.856 ms
chembl_14=#
chembl_14=# select * from get_mfp2_neighbors('Cc1ccc2nc(-
↪c3ccc(NC(C4N(C(c5cccs5)=O)CCC4)=O)cc3)sc2c1') limit 10;
 molregno |                              m                               |   ␣
↪similarity
----------+--------------------------------------------------------------+-------------
↪------
   472512 | Cc1ccc2nc(-c3ccc(NC(=O)C4CCN(C(=O)c5cccs5)CC4)cc3)sc2c1     | 0.
↪772727272727273
   471317 | Cc1ccc2nc(-c3ccc(NC(=O)C4CCCN(S(=O)(=O)c5cccs5)C4)cc3)sc2c1 | 0.
↪657534246575342
   471461 | Cc1ccc2nc(-c3ccc(NC(=O)C4CCN(S(=O)(=O)c5cccs5)CC4)cc3)sc2c1 | 0.
↪647887323943662
   471319 | Cc1ccc2nc(-c3ccc(NC(=O)C4CCN(S(=O)(=O)c5cccs5)C4)cc3)sc2c1  | 0.
↪638888888888889
  1032469 | O=C(Nc1nc2ccc(Cl)cc2s1)[C@@H]1CCCN1C(=O)c1cccs1             | 0.
↪623188405797101
   751668 | COc1ccc2nc(NC(=O)[C@@H]3CCCN3C(=O)c3cccs3)sc2c1             | 0.
↪619718309859155
   471318 | Cc1ccc2nc(-c3ccc(NC(=O)C4CN(S(=O)(=O)c5cccs5)C4)cc3)sc2c1   | 0.
↪611111111111111
   740754 | Cc1ccc(NC(=O)C2CCCN2C(=O)c2cccs2)cc1C                       | 0.
↪606060606060606
   732905 | O=C(Nc1ccc(S(=O)(=O)N2CCCC2)cc1)C1CCCN1C(=O)c1cccs1         | 0.
↪602941176470588
  1087495 | Cc1ccc(NC(=O)C2CCCN2C(=O)c2cccs2)c(C)c1                     | 0.
↪597014925373134
(10 rows)

Time: 5453.200 ms
chembl_14=# select * from get_mfp2_neighbors('Cc1ccc2nc(N(C)CC(=O)O)sc2c1') limit 10;
 molregno |                           m                           |    similarity
----------+-------------------------------------------------------+-------------------
   412312 | Cc1ccc2nc(N(C)CCN(C)c3nc4ccc(C)cc4s3)sc2c1            | 0.692307692307692
   470082 | CN(CC(=O)O)c1nc2cc([N+](=O)[O-])ccc2s1                | 0.583333333333333
  1040255 | CC(=O)N(CCCN(C)C)c1nc2ccc(C)cc2s1                     | 0.571428571428571
   773946 | Cl.CC(=O)N(CCCN(C)C)c1nc2ccc(C)cc2s1                  | 0.549019607843137
  1044892 | Cc1ccc2nc(N(CCN(C)C)C(=O)c3cc(Cl)sc3Cl)sc2c1          | 0.518518518518518
  1040496 | Cc1ccc2nc(N(CCCN(C)C)C(=O)CCc3ccccc3)sc2c1            | 0.517857142857143
  1049393 | Cc1ccc2nc(N(CCCN(C)C)C(=O)CS(=O)(=O)c3ccccc3)sc2c1    | 0.517857142857143
   441378 | Cc1ccc2nc(NC(=O)CCC(=O)O)sc2c1                        | 0.510204081632653
  1042958 | Cc1ccc2nc(N(CCN(C)C)C(=O)c3ccc4ccccc4c3)sc2c1          | 0.509090909090909
  1047691 | Cc1ccc(S(=O)(=O)CC(=O)N(CCCN(C)C)c2nc3ccc(C)cc3s2)cc1 | 0.509090909090909
(10 rows)

Time: 1797.656 ms
```

Adjusting the similarity cutoff

By default, the minimum similarity returned with a similarity search is 0.5. This can be adjusted with the rdkit.tanimoto_threshold (and rdkit.dice_threshold) configuration variables:

```
chembl_14=# select count(*) from get_mfp2_neighbors('Cc1ccc2nc(N(C)CC(=O)O)sc2c1');
 count
-------
    18
(1 row)

Time: 1199.751 ms
chembl_14=# set rdkit.tanimoto_threshold=0.7;
SET
Time: 0.191 ms
chembl_14=# select count(*) from get_mfp2_neighbors('Cc1ccc2nc(N(C)CC(=O)O)sc2c1');
 count
-------
     0
(1 row)

Time: 826.058 ms
chembl_14=# set rdkit.tanimoto_threshold=0.6;
SET
Time: 0.220 ms
chembl_14=# select count(*) from get_mfp2_neighbors('Cc1ccc2nc(N(C)CC(=O)O)sc2c1');
 count
-------
     1
(1 row)

Time: 1092.303 ms
chembl_14=# set rdkit.tanimoto_threshold=0.5
chembl_14-# ;
SET
Time: 0.257 ms
chembl_14=# select count(*) from get_mfp2_neighbors('Cc1ccc2nc(N(C)CC(=O)O)sc2c1');
 count
-------
    18
(1 row)

Time: 1081.721 ms
```

Using the MCS code

The most straightforward use of the MCS code is to find the maximum common substructure of a group of molecules:

```
chembl_20=# select fmcs(m::text) from rdk.mols join compound_records using (molregno)␣
↪where doc_id=3;                                                                      ␣
↪                      fmcs
------------------------------------------------------------------------------------
↪-----------------------------------------------------------------------------------
↪----------------
 [#6]1(-[#7](-[#6](-[#6]2:[#6]:[#6]:[#6](:[#6]:[#6]:2)-[#7]-[#6](-[#6]2:[#6](-[#6]3:[
↪#6]:[#6]:[#6]:[#6]:[#6]:3):[#6]:[#6]:[#6]:[#6]:2)=[#8])=[#8])-[#6]-[#6]-[#6]):[#6]:[
↪#16]:[#6]:[#6]:1
(1 row)

chembl_20=# select fmcs(m::text) from rdk.mols join compound_records using (molregno)␣
↪where doc_id=4;
                                  fmcs
------------------------------------------------------------------------
 [#6](-[#6]-,:[#6]-,:[#6]-,:[#6]-,:[#6])-[#7]-[#6]-[#6](-,:[#6])-,:[#6]
(1 row)
```

The same thing can be done with a SMILES column:

```
chembl_20=# select fmcs(canonical_smiles) from compound_structures join compound_
↪records using (molregno) where doc_id=4;
                                  fmcs
------------------------------------------------------------------------
 [#6](-[#7]-[#6]-[#6]-,:[#6]-,:[#6]-,:[#6]-,:[#6])-[#6](-,:[#6])-,:[#6]
(1 row)
```

It's also possible to adjust some of the parameters to the FMCS algorithm, though this is somewhat more painful as of this writing (the 2015_03_1 release). Here are a couple of examples:

```
chembl_20=# select fmcs_smiles(str,'{"Threshold":0.8}') from
chembl_20-#    (select string_agg(m::text,' ') as str from rdk.mols
chembl_20(#    join compound_records using (molregno) where doc_id=4) as str ;
                                                                           fmcs_smiles
------------------------------------------------------------------------------------
↪-----------------------------------------------------------------------------
 [#6]-[#6]-[#8]-[#6](-[#6](=[#8])-[#7]-[#6](-[#6])-[#6](-,:[#6])-,:[#6])-[#6](-[#8])-[
↪#6](-[#8])-[#6](-[#8]-[#6]-[#6])-[#6]-[#7]-[#6](-[#6])-[#6](-,:[#6])-,:[#6]
(1 row)

chembl_20=# select fmcs_smiles(str,'{"AtomCompare":"Any"}') from
chembl_20-# (select string_agg(m::text,' ') as str from rdk.mols
chembl_20(# join compound_records using (molregno) where doc_id=4) as str ;
                                                                              fmcs_
↪smiles
------------------------------------------------------------------------------------
↪-----------------------------------------------------------------------------------
 [#6]-,:[#6,#7]-[#8,#6]-[#6,#7](-[#6,#8]-[#7,#6]-,:[#6,#7]-,:[#6,#7]-,:[#7,#6]-,:[
↪#6])-[#6,#7]-[#6]-[#6](-[#8,#6]-[#6])-[#6,#7]-[#7,#6]-[#6]-,:[#6,#8]-,:[#7,#6]-,:[
↪#6]
(1 row)
```

Note The combination of `"AtomCompare":"Any"` and a value of `"Threshold"` that is less than 1.0 does a quite generic search and can results in very long search times. Using `"Timeout"` with this combination is recommended:

```
chembl_20=# select fmcs_smiles(str,'{"AtomCompare":"Any","CompleteRingsOnly":true,
↪"Threshold":0.8,"Timeout":60}') from
chembl_20-#   (select string_agg(m::text,' ') as str from rdk.mols
chembl_20(#   join compound_records using (molregno) where doc_id=3) as str ;
WARNING:  findMCS timed out, result is not maximal
                                                                                        ␣
↪    fmcs_smiles
-------------------------------------------------------------------------------------
↪------------------------------------------------------------------------------------
↪--------------------
 [#8]=[#6](-[#7]-[#6]1:[#6]:[#6]:[#6](:[#6]:[#6]:1)-[#6](=[#8])-[#7]1-[#6]-[#6]-[#6]-[
↪#6,#7]-[#6]2:[#6]-1:[#6]:[#6]:[#16]:2)-[#6]1:[#6]:[#6]:[#6]:[#6]:[#6]:1-[#6]1:[#6]:[
↪#6]:[#6]:[#6]:[#6]:1
(1 row)
```

Available parameters and their default values are:

- MaximizeBonds (true)
- Threshold (1.0)
- Timeout (-1, no timeout)
- MatchValences (false)
- MatchChiralTag (false) Applies to atoms
- RingMatchesRingOnly (false)
- CompleteRingsOnly (false)
- MatchStereo (false) Applies to bonds
- AtomCompare (“Elements”) can be “Elements”, “Isotopes”, or “Any”
- BondCompare (“Order”) can be “Order”, “OrderExact”, or “Any”

Reference Guide

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)

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.
- rdkit.do_chiral_sss : toggles whether or not stereochemistry is used in substructure matching. (*available from 2013_03 release*).

- rdkit.sss_fp_size : the size (in bits) of the fingerprint used for substructure screening.
- rdkit.morgan_fp_size : the size (in bits) of morgan fingerprints
- rdkit.featmorgan_fp_size : the size (in bits) of featmorgan fingerprints
- rdkit.layered_fp_size : the size (in bits) of layered fingerprints
- rdkit.rdkit_fp_size : the size (in bits) of RDKit fingerprints
- rdkit.torsion_fp_size : the size (in bits) of topological torsion bit vector fingerprints
- rdkit.atompair_fp_size : the size (in bits) of atom pair bit vector fingerprints
- rdkit.avalon_fp_size : the size (in bits) of avalon fingerprints

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.

Functions

Fingerprint Related

Generating fingerprints

- morgan_fp(mol,int default 2) : 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 default 2) : 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 default 2) : 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 default 2) : 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 fingerprint 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, bool default false) : returns a molecule for a CTAB (mol block) string, NULL if the molecule construction fails. The optional second argument controls whether or not the molecule's coordinates are saved.
- 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*)
- qmol_from_smiles(smiles) : returns a query molecule for a SMILES string, NULL if the molecule construction fails. Explicit Hs in the SMILES are converted into query features on the attached atom.
- qmol_from_ctab(ctab, bool default false) : returns a query molecule for a CTAB (mol block) string, NULL if the molecule construction fails. Explicit Hs in the SMILES are converted into query features on the attached atom. The optional second argument controls whether or not the molecule's coordinates are saved.
- 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*)
- mol_to_ctab(mol,bool default true) : returns a CTAB (mol block) string for a molecule. The optional second argument controls whether or not 2D coordinates will be generated for molecules that don't have coordinates. (*available from the 2014_03 release*)
- mol_to_svg(mol,string default '',int default 250, int default 200, string default '') : returns an SVG with a drawing of the molecule. The optional parameters are a string to use as the legend, the width of the image, the height of the image, and a JSON with additional rendering parameters. (*available from the 2016_09 release*)

Substructure operations

- substruct(mol,mol) : returns whether or not the second mol is a substructure of the first.
- substruct_count(mol,mol,bool default true) : returns the number of substructure matches between the second molecule and the first. The third argument toggles whether or not the matches are uniquified. (*available from 2013_03 release*)
- mol_adjust_query_properties(mol,string default '') : returns a new molecule with additional query information attached. (*available from the 2016_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*).
- mol_formula(mol,bool default false, bool default true) : returns a string with the molecular formula. The second argument controls whether isotope information is included in the formula; the third argument controls whether "D" and "T" are used instead of [2H] and [3H]. (*available from the 2014_03 release*)

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*).
- mol_numspiroatoms : returns the number of spiro atoms in a molecule (*available from 2015_09 release*).
- mol_numbridgeheadatoms : returns the number of bridgehead atoms in a molecule (*available from 2015_09 release*).

MCS

- fmcs(mols) : an aggregation function that calculates the MCS for a set of molecules
- fmcs_smiles(text, json default ‘’) : calculates the MCS for a space-separated set of SMILES. The optional json argument is used to provide parameters to the MCS code.

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.

Using the Cartridge from Python

The recommended adapter for connecting to postgresql is pyscopg2 (https://pypi.python.org/pypi/psycopg2).

Here’s an example of connecting to our local copy of ChEMBL and doing a basic substructure search:

```
>>> import psycopg2
>>> conn = psycopg2.connect(database='chembl_16')
>>> curs = conn.cursor()
>>> curs.execute('select * from rdk.mols where m@>%s',('c1cccc2c1nncc2',))
>>> curs.fetchone()
(9830, 'CC(C)Sc1ccc(CC2CCN(C3CCN(C(=O)c4cnnc5ccccc54)CC3)CC2)cc1')
```

That returns a SMILES for each molecule. If you plan to do more work with the molecules after retrieving them, it is much more efficient to ask postgresql to give you the molecules in pickled form:

```
>>> curs.execute('select molregno,mol_send(m) from rdk.mols where m@>%s',(
↪'c1cccc2c1nncc2',))
>>> row = curs.fetchone()
>>> row
(9830, <read-only buffer for 0x...>)
```

These pickles can then be converted into molecules:

```
>>> from rdkit import Chem
>>> m = Chem.Mol(str(row[1]))
>>> Chem.MolToSmiles(m,True)
'CC(C)Sc1ccc(CC2CCN(C3CCN(C(=O)c4cnnc5ccccc54)CC3)CC2)cc1'
```

License

CHAPTER 7

Additional Information

- Python API Documentation
- C++ API Documentation